Automated Signature Generation for Zero-day Polymorphic Worms Using a Double-honeynet

Mohssen M. Z. E. Mohammed

This thesis is submitted in partial fulfillment of the academic requirements

for the degree of

Doctor of Philosophy in Electrical Engineering

in the Faculty of Engineering and The Built Environment

University of Cape Town

February 2012

As the candidate's supervisor, I have approved this dissertation for submission.

Name: Professor H. Anthony Chan

Signed by candidate

Signature Removed

Signed: _____

Date: ____ February 10 2012_____

Declaration

I hereby declare that: (1) the above thesis is my own unaided work, both in conception and execution, and that apart from the normal guidance of my supervisor, I have received no assistance apart from that stated below; (2) except as stated below, neither the substance or any part of the thesis has been submitted in the past, or is being, or is to be submitted for a degree in the University or any other University.

I am now presenting the thesis for examination for the Degree of (PhD) in Electrical Engineering. I also grant the University free license to reproduce the above thesis in whole or in part, for the purpose of research.

Mohssen M. Z. E. Mohammed	February 9 2012
Name	Date

To my family

Abstract

Internet worms pose a major threat to Internet infrastructure security, and their destruction causes loss of millions of dollars. Therefore, the networks must be protected as much as possible to avoid losses. This thesis proposes an accurate system for signature generation for Zero-day polymorphic worms.

Thesis consists of two parts:

In part one, polymorphic worm instances are collected by designing a novel Double-honeynet system, which is able to detect new worms that have not been seen before. Unlimited honeynet outbound connections are introduced to collect all polymorphic worm instances. Therefore this system produces accurate worm signatures.

In part two, signatures are generated for the polymorphic worms that are collected by the Double-honeynet system. Both a Modified Knuth-Morris-Pratt (MKMP) Algorithm, which is string matching based, and a Modified Principal Component Analysis (MPCA), which is statistics based, are used. The MKMP algorithm compares the polymorphic worms substrings to find the multiple invariant substrings that are shared between all polymorphic worm instances and use them as signatures. The MPCA determines the most significant substrings that are shared between polymorphic worm instances and use them as signatures.

The experimental results show that the Modified Knuth-Morris-Pratt Algorithm and MPCA can successfully detect polymorphic worms with zero false positives and low false negatives.

Acknowledgements

First and foremost, I would like to thank the Almighty God for blessing invaluable gifts of health, strength, belief, love, hope, patience, and protection to me and my family throughout my study period. Had it not been the will of God, nothing would have been possible for me.

I am very much indebted to acknowledge my advisor, Prof. H. Anthony Chan for the outstanding encouragement, genuine guidance, constructive comments, and excellent cooperation, which enabled me to complete my study within the targeted time.

I pleasantly express my thanks to my Co-supervisor Mr. Neco Ventura for his support and valuable comments which allowed me to overcome many of the research hurdles.

The laboratory of the Communications Research Group (CRG) provided an ideal research environment for my PhD study at University of Cape Town.

I would like to thank my colleagues at the CRG lab for their ideas, efforts, and contributions which have significantly added value to my work.

I would like to thank Dr. Mohsin Hashim, Dr. Ihab Bashir, and Dr. Sakib Pathan for their support, and for many useful discussions.

I am extremely grateful to The Ministry of Higher Education and Scientific Research – The Republic of The Sudan for their generous financial support during this study.

Last, but not the least, I thank my parents and my sisters. It has been their genuine love and support that enabled me to reach this point.

Table of Contents

List of Figures

List of Tables

Glossary

Malware: Malware is a common name for all kinds of unwanted software such as viruses, worms, and Trojans.

Virus: A computer virus is a program that copies itself into (infects) other programs. It may or may not perform other tasks. It can be passed by infected files on a disk, by files downloaded from another computer, or by attachments sent via e-mail.

Worm: A computer worm is a program that spawns running copies of itself over a computer network and usually performs malicious actions, such as using up the computer's resources and possibly shutting the system down.

Polymorphic worm: A polymorphic worm is a computer worm that changes its appearance in every infection attempt.

Vulnerability: Vulnerabilities open security holes that can allow other applications to connect to the computer system without authorization or knowledge of a legitimate user.

Signature / Identifier: A signature is a set of characteristics that can identify a malware. Signatures are used by antivirus and antispyware products to determine if a file is malicious or not.

Honeypot: A honeypot is a security resource whose value lies in being probed, attacked, or compromised.

Honeynet: Honeynet is a network of standard production systems that are built together and are put behind some type of access control device (such as a firewall) to watch what happens to the traffic.

Sebek: Sebek is a data capture tool designed to capture all of the attacker's activities on a honypot, without the attacker knowing it.

Sebek Client: Sebek client is one of the sebek components that runs on the honeypots, its

purpose is to capture all ‘attacker's activities’ (Keystrokes, file uploads, passwords) then covertly send the data to the sebek server. Since the sebek client runs as a kernel module on the honeypots, it can capture all activities, including encrypted, such as SSH, IPSec.

Sebek Server: Sebek server is one of the sebek components which collects the data from the honeypots. The sebek server normally runs on the honeywall gateway.

DoS: A Denial-of-Service attack is an attempt to make a computer resource unavailable to its intended users.

IDS: Intrusion Detection System is software/hardware that detects and logs inappropriate, incorrect, or anomalous activity.

IPS: Intrusion Prevention System is defined as an in-line product that focuses on identifying and blocking malicious network activity in real time. Essentially, a combination of access control (Firewall/router) mechanisms is termed an intrusion detection system.

LAN: A Local Area Network (LAN) is a group of computers and associated devices that share a common communication line or wireless link.

Honeywall CDROM: The Honeywall CDROM is a bootable CD that installs onto a hard drive and comes with all the tools and functionality for you to implement data capture, control, and analysis for multiple honeypot deployments.

WAN: A Wide Area Network is a geographically dispersed telecommunications network. The term distinguishes a broader telecommunication structure from a local area network (LAN).

Chapter 1 Introduction

1.1 Problem Definition and Motivation for the Thesis

Computer Worm is a kind of malicious program that self-replicates automatically within a computer network. Worms are in general, a serious threat to computers connected to the Internet and its proper functioning. These malicious programs can spread by exploiting low-level software defects, and can use their victims for illegitimate activities; such as corrupting data, sending unsolicited electronic mail messages, generating traffic for distributed Denial of Service (DoS) attacks, or stealing information [2, 4]. Today the speed at which the worm propagates poses a serious security threat to the Internet. Polymorphic worm is a kind of worm that is able to change its payload in every infection attempt, so it can evade the Intrusion Detection Systems, and damage data, delay the network, cause information theft, and other illegal activities that lead to even for example, high financial loss. To defend the network against the worm, intrusion detection systems (IDS) such as Bro and Snort are commonly deployed at the edge of network and the Internet. The main principle of these IDSs is to analyze the traffic to compare it against the signatures stored in their databases. Whenever a novel worm is detected in the Internet, the common approach is that the experts from security community analyze the worm code manually, and produce a signature. The signature is then distributed and each IDS updates its database with this new signature.

This approach of creating signature is human intensive, very slow and when we have threats of very fast replicating worms (that take as small as few seconds to bring down the entire network) like zero day worms, the need of an alternative is recognized. The alternative approach is to find a way to automatically generate signatures that are relatively faster to generate and are of acceptable good quality. Our work is concerned to do an automated signature generation for zero-day polymorphic worms, so that we can secure the network of the universities, education institutes, companies, organizations, banks, etc. from this great danger.

1.2 Internet Worm Defense Methods

Due to the enormous threat from the worms, many efforts have been taken previously to tackle worms by detecting and preventing them. Later in this thesis, the relevant works are discussed, however in this section, internet worm defense methods and their limitations are mentioned in brief.

One avenue to deal with worms is prevention. We usually know that prevention is better than cure. Since worms need to exploit software defects, by eliminating all software defects we could eradicate worms. While theoretically this seems to be easy, the reality finds this as an almost impossible goal. Although, significant progress has been made on software development, testing, and verification, empirical evidence [62] suggests that we are still far from producing defect-free software.

Another avenue to solve the worm problem is containment. Containment systems accept that software has defects that can be exploited by worms, and they strive to contain a worm epidemic to a small fraction of the vulnerable machines. The main challenge in designing containment systems is that they need to be completely automatic, because worms can spread far faster than humans can respond [40, 63, 60]. Recent works on automatic containment [7, 8, 9, 64] have explored network-level approaches. These rely on heuristics to analyze network traffic and derive a packet classifier that blocks or rate-limits forwarding of worm packets.

It is hard to provide guarantees on the rate of false positives and false negatives with these approaches because there is no information about the software vulnerabilities exploited by worms at the network level. False negatives allow worms to escape containment, while false positives may cause network outages by blocking normal traffic. We believe that automatic containment systems will not be widely deployed unless they have a negligible false positive rate.

It should be noted here that dealing with the prevention mechanisms is out of the scope of this thesis because our work mainly focuses on containment mechanism of the worms.

1.3 Automated Zero-day Worm Detection

Bearing the difficulty of tackling worms in mind, in this section, we briefly discuss how our proposed Double-honeynet mechanism solves the existing limitations on the current worm detection systems.

We propose a Double-honeynet system to collect all polymorphic worm instances automatically without human interaction and to generate a signature based on the collected worm patterns. The Double-honeynet system is a hybrid system (Network-based and Host-based). The system operates at the Network-level by filtering unwanted traffic using Local Router and operates at the Host-level by allowing polymorphic worms to interact with Honeynet 1 and Honeynet 2 hosts. Interaction between the two honeynets works by forming a loop which allows us to collect all polymorphic worm instances. This mechanism reduces the false positives and false negatives dramatically which is the general limitation of the current worm detection systems.

1.4 Contributions to Automated Signature Generation Systems

The major contributions of this thesis are:

- Design of a novel Double-honeynet system, which is able to detect worms that are not seen before
- We introduce unlimited honeynet outbound connections that allow us to collect all polymorphic worm instances which enable our system to produce accurate worm signatures.
- Ability of the system to generate signatures to match all Polymorphic worm instances.
- The Double-honeynet system is a hybrid system with both Network-based and Host-based. This allows us to collect polymorphic worm instances on the network-level and host-level which reduces the false positives and false negatives dramatically.

These contributions are documented in the following selected peer reviewed publications:

1. Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, And Eihab Bashir, “Polymorphic Worms Detection Using A Supervised Machine Learning Technique”, will be appeared on The 2012 International Conference on Security and Management (SAM'12) Las Vegas, USA, July 2012, 16 – 19.

2. Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "Zero-day Polymophic Worms Detection Using a Modified Boyer-Moore Algorithm" Proceedings of the 2010 International Conference on Security and Management (SAM 2010), Las Vegas, USA, 12-15 July 2010.

3. Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim,Izzeldin Amin, "Accurate Signature Generation for Polymorphic Worms Using Principal Component Analysis," to appear in Proceedings of IEEE Globecom 2010 Workshop on Web and Pervasive Security (WPS 2010), Miami, Florida, USA,6-10 December 2010.

4. Mohssen M Z E Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, and Eihab Bashier, "Fast and Accurate Detection for Polymorphic Worms," The 5th International Conference for Internet Technology and Secured Transactions (ICITST-2010),London, UK.

5. Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "Polymorphic Worm Detection Using Double-Honeynet," Proceedings of The Fourth International Conference on Software Engineering Advances (ICSEA 2009), Porto, Portugal, 20-25 September 2009. IEEE Computer Society. ISBN 9780769537771.

6. Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "A modified Knuth-Morris-Pratt Algorithm for Zero-day Polymorphic Worms Detection," Proceedings of The 2009 International Conference on Security and Management (SAM'09), LasVegas, USA, 13-16 July

2009. IEEE. ISBN 9781601321244.

7. Mohssen Mohammed and H. Anthony Chan, "Honeycyber: Automated Signature Generation for Zero-day Polymorphic Worms," accepted at IEEE Military Communications Conference (MILCOM), San Diego, USA, 17-19 Nov. 2008. ISBN: 978-1-4244-2677-5.

8. Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, "Fast Automated Signature Generation for Polymorphic Worms Using Double-Honeynet," Proceedings of 3rd International Conference on Broadband Communications, Information Technology & Biomedical Applications (BroadCom 2008), Pretoria, South Africa, 23-26 November 2008.

9. Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, And Izzeldin Amin, " An Automated Signature Generation Approach for Polymorphic Worms Using Principal Component Analysis", the proceedings of the International Journal for Information Security Research (IJISR), Volume 1, Issue 1, March 2011, pp. 53-62 ,ISSN: 2042- 4639 (Online).

1.5 Organization of This Thesis

This thesis is organized as follows: after the introductory information in Chapter 1, Chapter 2 reviews worm attacks. Chapter 3 discusses prevention and detection approaches and literature review. Chapter 4 introduces the proposed Double-honeynet architecture to address the problems faced by current automated signature systems, and analysis of the outcomes and results. Signature generation algorithms for polymorphic worm and experimental implementation results are discussed in Chapter 5. Chapter 6 concludes the thesis and highlights the future work.

Chapter 2 Preliminaries of Worm and Worm Attacks

As noted earlier, *worms* are computer programs that self-replicate without requiring any human intervention, especially by sending copies of their code in network packets and ensuring the code is executed by the computers that receive it. When computers become infected, they spread further copies of the worm and possibly perform other malicious activities [2, 4].

2.1 Worm Infection

Remotely infecting a computer requires coercing the computer into running the worm code. To achieve this, worms exploit low-level software defects, also known as vulnerabilities. Vulnerabilities are common in current software, because today's software is usually large, complex, and mostly written in unsafe programming languages. Several different classes of vulnerabilities have been discovered over the years. Currently, buffer overflows, arithmetic overflows, memory management errors, and incorrect handling of format strings, are among the most common types of vulnerabilities exploitable by worms [40].

While we should expect new types of vulnerabilities to be discovered in the future, the mechanisms used by worms to gain control of a program's execution should change less frequently. Currently, worms gain control of the execution of a remote program using one of three mechanisms: injecting new code into the program, injecting new control-flow edges into the program (e.g., forcing the program to call functions that shouldn't be called), and corrupting data used by the program.

2.2 Spread of Internet Worms

After infecting a computer, worms typically use it to infect other computers, giving rise to a propagation process which has many similarities with the spread of human diseases.

The spread of the worm in its most basic sense depends most greatly on how it chooses its victims. This not only affects the spread and pace of the worm network, but also its survivability and persistence as cleanup efforts begin. Classically, worms have used random walks of the

Internet to find hosts and attack. However, new attack models have emerged that demonstrate increased aggressiveness [2, 4].

2.2.1 Random Scanning

The simplest way for a worm to spread as far as it can is to use random network scanning. In this method, the worm node randomly generates a network to scan. This worm node then begins to search for potential victims in that network space and attacks vulnerable hosts. This random walk is the classic spread model for network-based worms.

2.2.2 Random Scanning using lists

In this method, the worm carries a list of numbers used to assist in the generation of the networks to probe and attack. This list is built from assigned and used address space from the Internet. By using this approach, the worm is able to focus on locations where hosts are likely to be present, improving the worm's efficiency.

2.2.3 Island Hopping

The third type of network scanning that worms perform is typically called island hopping. This is so named because it treats network blocks as islands on which it focuses attention before hopping away to a new, random destination.

2.2.4 Directed Attacking

Another targeting and direction method that can be used by a worm is that of directing its attack at a particular network. In this scenario, a worm carries a target network to penetrate and focuses its efforts on that network. This type of worm attack would be used in information warfare.

2.2.5 Hit-list Scanning

Hit list contains the addresses and information of nodes vulnerable to the worm's attacks. This list is generated from scans made before unleashing the worm. For example, an attacker

would scan the Internet to find 50,000 hosts vulnerable to a particular Web server exploit. This list is carried by the worm as it progresses, and is used to direct its attack. When a node is attacked and compromised, the hit list splits into half and one-half remains with the parent node and the other half goes to the child node. This mechanism continues and the worm's efficiency improves with every permutation.

2.3 Worm Components

There are five basic components of worm [2, 4]:

2.3.1 Reconnaissance

The worm network has to hunt out other network nodes to infect. This component of the worm is responsible for discovering hosts on the network that are capable of being compromised by the worm's known methods.

2.3.2 Attack Components

These are used to launch an attack against an identified target system. Attacks can include the traditional buffer or heap overflow, string formatting attacks, Unicode misinterpretations (in the case of IIS attacks), and misconfigurations.

2.3.3 Communication Components

Nodes in the worm network can talk to each other. The communication components give the worms the interface to send messages between nodes or some other central location.

2.3.4 Command Components

Once compromised, the nodes in the worm network can be issued operation commands using this component. The command element provides the interface to the worm node to issue and act on commands.

2.3.5 Intelligence Components

To communicate effectively, the worm network needs to know the location of the nodes as well as characteristics about them. The intelligence portion of the worm network provides the information needed to be able to contact with other worm nodes, which can be accomplished in a variety of ways.

2.4 Worm Examples

In this section, we give examples of Internet worms [2, 3, 4]. These examples contain one of the first computer worms distributed via the Internet which is known as Morris worm, then we show which vulnerabilities and operating systems the worm would target, and the high speed of worms spreading in the network and infecting computers. These examples also show instances of polymorphic worms.

2.4.1 Morris Worm

On the evening of November 2, 1988, a self-replicating program was released to attack the Internet. The program, later called the Morris worm, invaded VAX and Sun-3 computers running versions of Berkeley UNIX and used their resources to attack more computers. Within the space of hours this program had spread across the United States, infecting thousands of computers and making many of them unusable due to the burden of its activity. Although the worm was designed to spread itself to as many computers as possible and took only a tiny process to be unnoticeable, it did work strikingly due to mistakenly underestimating its spreading power and overload. As time passed by, some of these affected machines became so loaded with running processes (because they were repeatedly affected) that they were unable to continue any processing. Some machines failed completely when their swap space or process tables were exhausted.

2.4.2 Melissa

The Melissa Worm was first recognized on 26th March 1999, it was the first major mail worm - a form of worm which was to become hugely prevalent.

Melissa contained a Word macro virus (Macro viruses are computer viruses that use an application's own macro programming language to distribute themselves). This particular type of worm could spread in a semi-active manner. It attacked Microsoft's Outlook and Word programs (Any time an infected user attached a Word document with an email, this email would be sent to the first 50 addresses in the recipients' address book, if they had used Outlook as the email client).

2.4.3 Sadmind

A On May 8, 2001, a self-propagating malicious worm, Sadmind/IIS, was created. The worm uses two vulnerabilities to compromise systems and deface Web pages. It affects systems running unpatched Microsoft Internet Information Server (IIS) and systems running unpatched Solaris up to version 7. Intruders can use the vulnerabilities exploited by this worm to execute arbitrary code with root privileges on vulnerable Solaris systems and arbitrary commands with the privileges of the IUSR machinename account on vulnerable Windows systems.

2.4.4 Code Red I and Code Red II

In 2001, two worms exploited the same vulnerability to disturb the Internet: Code Red I on July 19 and Code Red II on August. Both of them exploited the vulnerability of buffer overflow bugs in Microsoft IIS Indexing Service DLL. They affected Microsoft Windows NT 4.0 with IIS 4.0 or IIS 5.0 enabled and Index Server 2.0 installed, Windows 2000 with IIS 4.0 or IIS 5.0 enabled and Indexing services installed and other systems running IIS. More than 250,000 hosts suffered from their attacks.

2.4.5 Nimda

On September 18, 2001, a worm, named W32/Nimda or Concept Virus (CV) v.5, propagated in the Internet to attack systems running Microsoft Windows 95, 98, ME, NT,and 2000. With the worm, intruders can execute arbitrary commands within the LocalSystem security context on machines running unpatched versions of IIS. In the case in which a client is compromised, the worm will run with the same privileges as the user who triggered it. The infected computers may suffer from DoS (Denial of Service) caused by network scanning and e-mail propagation.

2.4.6 SQL Slammer

On January 25 2003, a worm referred to as SQL Slammer, W32.Slammer, or Sapphire caused varied levels of network performance degradation across the Internet. This worm affects Microsoft SQL Server 2000 and Microsoft Desktop Engine (MSDE) 2000. The high volume of user datagram protocol (UDP) traffic generated by the infected hosts may lead to performance degradation against Internet-connected hosts or those computers that stay on the same network of a compromised host. The worm exploits the vulnerability of stack buffer overflow in the Resolution Service of Microsoft SQL Server 2000 and MSDE 2000 so that an intruder can execute arbitrary code with the same privileges as the SQL server.

2.4.7 Blaster Worm

On August 11, 2003, a worm, named Blaster, was launched. It affects computers running Microsoft Windows NT 4.0, Microsoft Windows 2000, Microsoft Windows XP, and Microsoft Windows Server 2003. This worm exploits vulnerability in the Microsoft Remote Procedure Call (RPC) interface. This vulnerability affects a distributed component object model (DCOM) interface with RPC, which listens on TCP/IP port 135. This interface handles the DCOM object activation requests that are sent by client machines to the server. Due to incorrect handling of malformed messages exchanged over TCP/IP, an attacker can use buffer overflow to execute arbitrary code with system privileges or cause denial of service.

2.4.8 Sasser Worm

Sasser worm is a network worm that was first detected in April 2004. Sasser worm exploits buffer overflow vulnerability in the Windows Local Security Authority Service Server (LSASS) on TCP port 445. The vulnerability allows a remote attacker to execute arbitrary code with system privileges.

2.4.9 Conficker Worm

Conficker is a computer worm that targets the Microsoft Windows operating system which was first detected in November 2008. Conficker worm exploit a specific vulnerability in the

Server Service on Windows computers, in which an already-infected source computer uses a specially-crafted RPC request to force a buffer overflow and executes shellcode on the target computer. On the source computer, the worm runs an HTTP server on a port between 1024 and 10000; the target shellcode connects back to this HTTP server to download a copy of the worm in DLL form, which is then attached to svchost.exe.

2.4.10 Allaple Worm

Allaple worm is a network worm designed for the Windows platform that was first detected in August 2008. Once it is executed, Allaple will search local disks for HTML files and inject code into them to activate the installed copy of it. Some variants of Allaple may spread to other network computers by exploiting common buffer overflow vulnerabilities, including: SRVSVC (MS06-040), RPC-DCOM (MS04-012), PNP (MS05-039) and ASN.1 (MS04-007) and by copying itself to network shares protected by weak passwords.

2.5 Polymorphic Worms: Definition and Anatomy

2.5.1 Polymorphic Worm Definition

A polymorphic worm is a computer worm that changes its appearance in every infection attempt [2, 3, 4].

2.5.2 Polymorphic Worm Structure

As stated in [66], in a sample of polymorphic worm, we can identify the following components:

Protocol framework. To infect new hosts and continue their spread, worms have to exploit a given vulnerability. This vulnerability, in many cases, is associated with a particular application code and execution path in this code. This execution path can be activated by few, or much often one type of particular protocol request.

Exploit bytes. These bytes are used by the worm to exploit the vulnerability. They are necessary for the correct execution of the attack.

Worm body. These bytes contain instructions executed by the worm instances on new infected victims. In polymorphic worms, these bytes can assume different values in each instance.

Polymorphic decryptor. The polymorphic decryptor decodes the worm body and starts its execution.

Others bytes. These bytes do not affect the successful execution of both the worm body and exploit bytes.

2.5.3 Invariant Bytes

In a polymorphic worm sample, we can classify three kinds of bytes: invariant, code, and wildcard [66].

Invariant bytes are those with a fixed value in every possible instance. If their values are changed, the exploit no longer could work. They can be part of the protocol framework and exploit bytes but in some cases, also of the worm body or the polymorphic decryptor. Such bytes are very useful in signature generation because they are absolutely necessary for the exploit to work and their content is replicated across worm instances. **Code bytes** come from components like the worm body or decryption routine, in which there are instructions to be executed. Although code section of worm samples can be subjected to polymorphism and encryption techniques, and thus they can assume different shapes in each instance, polymorphic engines are not perfect and some of these bytes can present invariant values. Lastly, **wildcard bytes** are bytes that may take any value without affecting worms' spreading capabilities.

2.5.4 Signature Classes for Polymorphic Worms

Signatures for polymorphic worms can be classified into two broad categories: Content-based signatures that aim at using similarity in different instances of byte sequences to characterize a given worm, and Behavior-based signatures that aim at characterizing worms through understanding the semantics of their byte sequences. Our work focuses on Content-based signatures. An advantage of Content-based signatures is that they allow us to treat the worms as strings of bytes and do not depend upon any protocol or server information. In addition, the

Content-based signatures can easily be incorporated into firewalls or NIDSs.

2.5.5 Polymorphic Worm Techniques

The attackers will try every possible way to extend the life time of Internet worms. In order to evade the signature-based system, a polymorphic worm appears differently each time it replicates itself. This section discusses the polymorphism of Internet worms and polymorphic worm anatomy.

There are many ways to make polymorphic worms [12, 55, 56, 58, 59]. One technique relies on self encryption with a variable key. It encrypts the body of a worm, which erases both signatures and statistical characteristics of the worm byte string. A copy of the worm, the decryption routine, and the key are sent to a victim machine, where the encrypted text is turned into a regular worm program by the decryption routine. The program is then executed to infect other victims and possibly damage the local system. If the same decryption routine is always used, the byte sequence in the decryption routine can serve as the worm signature.

A more sophisticated method of polymorphism is to change the decryption routine each time a copy of the worm is sent to another victim host. This can be achieved by keeping several decryption routines in a worm. When the worm tries to make a copy, one routine is randomly selected and other routines are encrypted together with the worm body. The number of different decryption routines is limited by the total length of the worm. Given a limited number of decryption routines, it is possible to identify all of them as attack signatures after enough samples of the worm have been obtained.

Another polymorphism technique is called garbage-code insertion. It inserts garbage instructions into the copies of a worm. For example, a number of *nop* (i.e., no operation) instructions can be inserted into different places of the worm body, thus making it more difficult to compare the byte sequences of two instances of the same worm. However, from the statistics point of view, the frequencies of the garbage instructions in a worm can differ greatly from those in normal traffic. If that is the case, anomaly-detection systems can be used to detect the worm. Furthermore, some garbage instructions such as *nop* can be easily identified and removed. For

better obfuscated garbage, techniques of executable analysis can be used to identify and remove those instructions that will never be executed.

The instruction-substitution technique replaces one instruction sequence with a different but equivalent sequence. Unless the substitution is done over the entire code without compromising the code integrity (which is a great challenge by itself), it is likely that shorter signatures can be identified from the stationary portion of the worm. The code-transposition technique changes the order of the instructions with the help of jumps. The excess jump instructions provide a statistical clue, and executable-analysis techniques can help remove the unnecessary jump instructions. Finally, the register-reassignment technique swaps the usage of the registers, which causes extensive "minor" changes in the code sequence.

2.6 Chapter summary

In this chapter, we have discussed how a worm infects a computer in a network and starts spreading to infect other computers in the network using worm components collaboration. We have mentioned some examples of worms and the vulnerabilities that they exploit. We looked at the structure of polymorphic worm which helps infection and spreading in a network. Moreover, the chapter described the techniques that polymorphic worms use to change their payload in every infection attempt.

Chapter 3 Prevention and Detection Approaches and Literature Review

This chapter contains two parts. The first part discusses the approaches that are used to prevent and detect the Internet worms. In addition, we discuss the attack detection technologies such as Intrusion Detection System (IDS), Firewalls, and Honeypots. The second part discusses the related works regarding automated signature generation systems for polymorphic worms.

3.1 Prevention and Detection of Worms

There are several sub-problems to the problem of worms. A worm is after all a program that remotely exploits some vulnerability in some application and hijacks the control flow of that application. So, the genesis of the problem is in the vulnerability that can be remotely exploited. Thus, **prevention** of such vulnerabilities comes first and then the attacks that exploit them form the first problem to deal with.

However, there is a large legacy of programs already in use that cannot be discarded overnight or cannot be relieved of such vulnerabilities easily. Given that there are also several undiscovered vulnerabilities in existing programs, it is fair to assume that exploits will be written for them by attackers who find them. So, **detection** of these attacks forms the second problem to be addressed.

3.1.1 Prevention

There are two different approaches to prevent worm attacks. One is to prevent vulnerabilities. Second one is to prevent exploitation of vulnerabilities. Such prevention not only guards against worm attacks but also intrusions of any kind [4].

3.1.1.1 Prevention of vulnerabilities

Secure Programming languages and practices: Most, not all, vulnerabilities can be avoided by good programming practices and secure design of protocols and software architectures. No matter how good software systems are, untenable assumptions and betrayed

trusts will make them vulnerable. Protocols and software architectures can be proved or verified by theorem provers such as HOL [19] but there is always a chance for human error and carelessness even in the most careful of programmers. Also, C [21], the most common language with which critical applications are programmed due to the efficiency and low-level control of data structures and memory that it offers, does not inherently offer safe and secure constructs. Vulnerabilities such as buffer overflows in C programs are possible, though caused by human-errors, because it is legitimate to write beyond the array and string boundaries in C. Thus, there is a need for more secure programming and execution environments. Fortunately, help is available for securing programs in the form of:

1. *Static analysis* tools which identify programming constructs in general that can lead to vulnerabilities. Lint is one of the most popular tools of such kind. LCLint [22, 23], is another one. MOPS [24, 25] is a model checking tool to examine source code for conformity to certain security properties. These properties are expressed as predicates and the tool uses model-checking to verify conformation. Metal [26, 27], and SLAM [28] are two more examples of such types of tools.

2. *Run-time checking* of program status by the use of assert statements in C, but they are usually turned off in the production versions of the software to avoid performance degradation [29].

3. A combination of both of the above. Systems such as CCured [30] perform static analysis and automatically insert run-time checks where safety cannot be guaranteed statically. These systems can also be used to retort legacy C code to prevent vulnerabilities.

4. Safe Languages offer the most promise. These languages such as Java and Cyclone [29] offer no scope for vulnerabilities. Cyclone, a dialect of C, ensures this by enforcing safe programming practices - it refuses to compile unsafe programs such as those that use uninitialized pointers; revoking some of the privileges such as unsafe casts, setjmp, longjmp, implicit returns, etc., that were available to C programmers; and by following the third technique mentioned above - a

combination of static analysis and inserting run-time checkers or assertions.

However, Java's type-checking system can itself be attacked exposing Java programs and Java virtual machines to danger [67]. Moreover, high level languages such as Java do not provide the low-level control that C provides. Whereas, Cyclone, provides a safer programming environment by a combination of static-analysis and inserting run-time checks, yet maintaining the low-level of control that C offers to the programmers.

Secure execution environments: A secure execution environment can also make sure that there are no vulnerabilities. A straightforward approach to provide a secure execution environment is to instrument each memory access with assertions for memory integrity.

Purify [32] is a tool that adopts this approach for C programs. However, it has a high performance penalty that prevents it from being used in the production environment. It can however be used as a debugger.

3.1.1.2 Prevention of exploits

Though a long list of mechanisms are available for prevention of vulnerabilities, no single tool's (or mechanism's) coverage is complete. Moreover, some of the tools are hard to use or have severe performance penalties and hence, are not used in production environments. Therefore, software continues to be shipped with vulnerabilities and attackers continue to write exploits. Even if all future systems ship without any vulnerability, there is a huge legacy of systems with vulnerabilities. Preventing exploits of those vulnerabilities, both known and unknown, is thus convenient. There are several perspectives from which this is achieved.

1. *Access Control Matrix and Lists (OS Perspective):* Traditionally, the responsibility for preventing mischief, data theft, accidents and deliberate vandalism and maintaining the integrity of computer systems has been taken up by the operating system. This responsibility is satisfied by controlling access to resources as dictated by the Access Control Matrix [33, 34]. Each entry in this matrix specifies the set of access rights to a resource that a process gets when executing in a certain protection domain. On timesharing multi-user systems such as UNIX, protection

domains are defined to be users and the Access Control Matrix is implemented as an Access Control List. This is in addition to the regular UNIX file permissions based on user groups, thus allowing arbitrary subsets of users and groups [36].

2. *Firewalls and IPS (Network Perspectives)* - Another way to prevent exploits is to filter exploit traffic at the network level based on certain rules and policies. Such traffic filtering is implemented mostly at the border gateways of networks and sometimes at the network layer of the network protocol stack on individual machines. An example policy is to never accept any TCP connection from a particular IP address. Another example may be to drop connections whose packet contents match of a certain pattern. The former is usually enforced by some kind of software called a firewall; for example, netfilters' iptables [38]. The latter is enforced by Intrusion Prevention Systems based on signatures; as an example, Snort-inline. There is another class of closely related software called Intrusion Detection Systems, which we will talk about shortly.

3. *Deterrents (Legal Perspective):* Several technical and legal measures have been undertaken to deter mischief mongers from tampering with computer systems. Enactment and enforcement of laws in combination with building up of audit trails [37] on computers (to serve incriminating evidence) have contributed to a great extent to securing computers.

3.2 Intrusion Detection Systems (IDSs)

The research community has proposed and built intrusion detection systems (IDSs) to defend against Internet worms (and other attacks) [53, 54]. Intrusion detection is the process of monitoring computers or networks for unauthorized entrance, activity, or file modification. IDS can also be used to monitor network traffic, thereby detecting if a system is being targeted by a network attack such as a denial of service attack.

There are two basic types of intrusion detection: **host-based and network-based**. **Host-based** IDSs examine data held on individual computers that serve as hosts, while **network-based**

IDSs examine data exchanged between computers.

There are two basic techniques used to detect intruders: **Anomaly Detection and Misuse Detection** (Signature Detection). **Anomaly Detection** is designed to uncover abnormal patterns of behavior, the IDS establishes a baseline of normal usage patterns, and anything that widely deviates from it gets flagged as a possible intrusion. Although these systems can detect previously unknown attacks, they have high false positives when the normal activities are diverse and unpredictable. **Misuse detection** (signature detection), which is commonly called Signature Detection uses specifically known patterns of unauthorized behavior to predict and detect subsequent similar attempts. These specific patterns are called signatures. They can detect the known worms but will fail on the new types.

Most deployed worm-detection systems are signature-based, which belongs to the Misuse detection category. They look for specific byte sequences (called attack signatures) that are known to appear in the attack traffic. The signatures are manually identified by human experts through careful analysis of the byte sequence from captured attack traffic. A good signature should be one that consistently shows up in attack traffic but rarely appears in normal traffic.

3.3 Firewalls

Firewall is a system designed to prevent unauthorized access to or from a private network. Firewalls can be implemented in both hardware and software, or a combination of both. Firewalls are frequently used to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets. All messages entering or leaving the intranet pass through the firewall, which examines each message and blocks those that do not meet the specified security criteria [1, 3].

There are several types of firewall techniques:

- Packet filter: Looks at each packet entering or leaving the network and accepts or rejects it based on user-defined rules. Packet filtering is fairly effective and transparent to users, but it is difficult to configure. In addition, it is susceptible to

IP spoofing.

- Application gateway: Applies security mechanisms to specific applications, such as FTP and Telnet servers. This is very effective, but can impose performance degradation.

- Circuit-level gateway: Applies security mechanisms when a TCP or UDP connection is established. Once the connection has been made, packets can flow between the hosts without further checking.

- Proxy server: Intercepts all messages entering and leaving the network. The proxy server effectively hides the true network addresses.

3.4 Honeypots

A honeypot is a security resource whose value lies in being probed, attacked, or compromised. This means that whatever we designate as a honeypot, our expectations and goals are to have the system probed, attacked, and potentially exploited. It does not matter what the resource is (a router, scripts running emulated services, a jail, an actual production system). What does matter is that the resource's value lies in its being attacked. If the system is never probed or attacked, then it has little or no value. This is the exact opposite of most of the production systems, which we usually do not want to be probed or attacked [1].

3.4.1 The Value of Honeypots

Honeypots are a highly flexible technology that can be applied to a variety of situations. As security tools, they have specific advantages. Specifically, honeypots collect small amounts of data, but most of this is information of high value. They have the ability to effectively work in resource intensive environments, and conceptually they are very simple devices. Also, they quickly demonstrate their value by detecting and capturing unauthorized activity.

However, honeypots share several major disadvantages. The most critical is that they have a narrow field of view. If they are not attacked, they have no value. Second, certain honeypots

can be fingerprinted, making detection possible. The third disadvantage is that honeypots can add additional risk: The honeypot may be used to attack or harm other systems or organizations. Any time we add additional services or applications to our environment, there are more things that can go wrong.

Within the three areas of security—prevention, detection and response—the primary value of production honeypots is detection. Because, production honeypots greatly reduce the problem of both false negatives and false positives; they make a highly efficient technology for detecting unauthorized activity. They also have some value with respect to reaction and, in relation to this, helping organizations to develop their incident response skills. For prevention purposes, production honeypots are of minimal value. The concepts of deception and deterrence can be applied with honeypots to prevent attacks, but most of the organizations are better off spending their limited resources on security best practices, such as patching vulnerable services. Honeypots will not stop vulnerable systems from being hacked.

Research honeypots do not mitigate risk, but they primarily are used to gain information about threats. This information is then used to better understand and protect against these threats. When deploying honeypots, it is critical that organizations have a clearly defined security policy stating what activity is and is not authorized, including the use of honeypots to detect and monitor [1].

3.4.2 Honeypots types

Level of interaction gives us a scale with which we could measure and compare honeypots. The more a honeypot can do and the more an attacker can do to a honeypot, the greater the information that can be derived from it. However, by the same token, the more an attacker can do to the honeypot, the more potential damage an attacker can incur. Honeypots fall into three categories which are Low-Interaction Honeypots, Mid-Interaction Honeypots, and High-Interaction Honeypots [1].

3.4.2.1 Low-Interaction Honeypots

Low-interaction honeypots typically are the easiest to install, configure, deploy, and

maintain because of their simple design and basic functionality. Normally, these technologies merely emulate a variety of services. The attacker is limited to interacting with these pre-designated services. For example, a low-interaction honeypot could emulate a standard Unix server with several running services, such as Telnet and FTP. An attacker could Telnet to the honeypot, get a banner that states the operating system, and perhaps obtain a login prompt. The attacker can then attempt to login by brute force method or by guessing the passwords. The honeypot would capture and collect these attempts, but there is no real operating system for the attacker to log on to. The attacker's interaction is limited to login attempts.

3.4.2.2 Mid-Interaction Honeypots

Medium-interaction honeypots offer attackers more ability to interact than the low-interaction honeypots do, but have less functionality than those of the high-interaction solutions. They can expect certain activity and are designed to give certain responses beyond what a low-interaction honeypot would give. For example, perhaps there is a worm scanning for specific IIS vulnerabilities. A honeypot could be built to imitate a Microsoft IIS Web server, including the additional functionality that normally accompanies the application. The emulated IIS Web server could then be customized to present whatever functionality or behavior the specific worm is looking for. Whenever an HTTP connection is made to the honeypot, it would respond as an IIS Web server, giving the attacker the opportunity to interact with actual IIS functionality. This level of interaction is greater than the low-level honeypot, which would have most likely simply presented an HTTP banner. In the case of the worm, our intent is for it to attack the honeypot so we can capture the worm payload for future analysis. However, the worm has not been given a full operating system with which to interact, limiting risk. There is only an emulated application. As such, this would not be a high level of interaction.

3.4.2.3 High-Interaction Honeypots

High-interaction honeypots are the highest levels of honeypot technologies. They give us a vast amount of information about attackers, but they are extremely time consuming to build and maintain, and also they come with the highest level of risk. The goal of a high-interaction honeypot is to give the attacker access to a real operating system where nothing is emulated or restricted. We can discover new tools, identify new vulnerabilities in operating systems or

applications, and learn how blackhats communicate among one other. The possibilities are almost limitless, making high-interaction honeypots an extremely powerful weapon.

3.4.3 Honeynet

Honeynets are high-interaction honeypots. In fact, it is difficult to envisage any other honeypot solution that can offer a greater level of interaction than honeynets do. The concept of a Honeynet is simple: Building a network of standard production systems, just as we would find in most organizations today. Putting this network of systems behind some type of access control device (such as a firewall) and watching what happens. Attackers can probe, attack, and exploit any system within the Honeynet, giving them full operating systems and applications to interact with. No services are emulated, and no caged environments are created. The systems within a Honeynet can be anything: a Solaris server running an Oracle database, a Windows XP server running an IIS Web server, a Cisco router, etc. In short, the systems within a Honeynet are true production systems [1].

3.4.4 Virtual Honeynets

Virtual Honeynets represent a relatively new field for Honeynets. The concept is to virtually run an entire Honeynet on a single, physical system. The purpose of this is to make Honeynets a cheaper solution that is easier to manage. Instead of investing in large amounts of hardware, all of the hardware requirements are combined into a single system [1].

3.5 Related Works Regarding Automated Signature Generation Systems

Previously proposed techniques to mitigate worm attacks can be divided into network-based and host-based mechanisms. Network-based mechanisms exclusively analyze network traffic, while host-based systems use information available at the end-hosts. This chapter discusses previous proposals in each of these areas.

3.5.1 Network-based mechanisms

One of the first systems proposed was Honeycomb developed by Kreibich and Crowcroft. Honeycomb generates signatures from traffic observed at a honeypot via its implementation as a Honeyd [7] plugin. The longest common substring (LCS) algorithm, which looks for the longest shared byte sequences across pairs of connections, is at the heart of Honeycomb. Honeycomb generates signatures consisting of a single, contiguous substring of a worm's payload to match all worm instances.

Kim and Karp [8] described the Autograph system for automated generation of signatures to detect worms. Unlike Honeycomb, Autograph's inputs are packet traces from a DMZ (Demilitarized Zone) that includes benign traffic. Content blocks that match "enough" suspicious flows are used as inputs to COPP, an algorithm based on Rabin fingerprints that searches for repeated byte sequences by partitioning the payload into content blocks. Similar to Honeycomb, Autograph generates signatures consisting of a single, contiguous substring of a worm's payload to match all worm instances.

S. Singh, C. Estan, G. Varghese, and S. Savage [9] described the Earlybird system for generating signatures to detect worms. This system measures packet-content prevalence at a single monitoring point such as a network DMZ. By counting the number of distinct sources and destinations associated with strings that repeat often in the payload, Earlybird distinguishes benign repetitions from epidemic content. Earlybird, also like Honeycomb and Autograph, generates signatures consisting of a single, contiguous substring of a worm's payload to match all worm instances.

The above systems generate a single signature to match all worm instances based on the assumption that there exists a single payload substring that will remain invariant across worm connections. Single signature is not qualified enough to match all worm instances with low false positives and low false negatives. Our polymorphic worm signature is based on the idea that there are more than one substring(s) that are shared between all polymorphic worm instances.

There are also some content-based systems like Polygraph, Hamsa, and LISABETH [10,

13, 14] that have been deployed. All these systems, similar to our system, generate automated signatures for polymorphic worms based on the following fact: there are multiple invariant substrings that must often be present in all variants of polymorphic worm payloads, even if the payload changes in every infection. All these systems capture the packet payloads at the network level. Polymorphic worm does not change its payload till it reaches the host level, so these systems cannot capture the remaining instances of the polymorphic worm. However, the systems may capture different types of polymorphic worms where each of them exploits a different vulnerability from each other. So, in this case, it may be difficult for these systems to find invariant contents shared between these polymorphic worms, because they exploit different vulnerabilities. We propose a Double-honeynet system to capture all polymorphic worm instances automatically without human interaction. The interactions between the two honeynets allow us to capture the remaining instances of the polymorphic worm.

An Architecture for Generating Semantics-Aware Signatures by Yegneswaran, J. Giffin, P. Barford, and S. Jha [11] described Nemean. Nemean incorporates protocol semantics into the signature generation algorithm. By doing so, it is able to handle a broader class of attacks. The coverage of Nemean is wide, which makes us believe that our system is better in dealing with polymorphic worms especially.

An Automated Signature-Based Approach against Polymorphic Internet Worms by Yong Tang and Shigang Chen [12] described a system to detect new worms and generate signatures automatically. This system implemented a double-honeypots (inbound honeypot and outbound honeypot) to capture worms payloads. The inbound honeypot is implemented as a high-interaction honeypot, whereas the outbound honeypot is implemented as a low-interaction honeypot. This system has some limitations. The outbound honeypot is not able to make outbound connections because it is implemented as low-interaction honeypot which is not able to capture the remaining instances of the polymorphic worm. Our system overcomes this disadvantage by using a Double-honeynet system (high-interaction honeypot), which enables us to make unlimited outbound connections between them, so we can capture the remaining instances of the polymorphic worm.

3.5.2 Host-based mechanisms

TaintCheck [41] and Vigilante [40] benefit from knowledge about the state of the host during an attack. They perform dynamic dataflow analysis to track program activity, including buffer overflows and control transfers. They identify worm signatures in terms of program behavior rather than packet payload.

Buttercup [39] proposed identifying the return address range used in worm attack messages and filtering messages that include such addresses. To reduce false positives, their system searches for the return address value starting at a predetermined offset in messages, and stops after a configurable number of bytes have been checked. While Buttercup requires these addresses to be externally specified, TaintCheck [41] proposed to obtain them automatically, by using the exact return address observed in attack messages. These systems can have false positives, because the 4 byte sequences used as a return address can appear in normal messages. The system can also have false negatives, since attackers can use a wide range of values of return addresses, by searching the address space of vulnerable applications for sequences of bytes that correspond to instructions that transfer control to the worm code.

ARBOR [18] generates signatures based on the size of network messages and the fraction of non-ASCII characters in them. Its signatures also include host context: messages are dropped at specific code locations, and when specific call sequences are observed. ARBOR can still have false positives and false negatives.

COVERS [17] also generates signatures based on length of inputs and fraction of non-ASCII characters in them, but includes an input correlation mechanism to identify attack packets and the specific bytes in those packets that were involved in an observed security fault. COVERS does not provide guarantees on the rate of false positives or false negatives.

There is no approach used in the above motioned systems to collect all polymorphic worm instances. Therefore, signatures produced by these systems suffer from high false positives. The Double-honeynet system provides a solution to collect all polymorphic worm instances. Interaction between the two honeynets works by forming a loop which allows us to collect all

polymorphic worm instances. This mechanism reduces the false positives and false negatives dramatically.

3.6 Chapter summary

In this chapter, we have discussed the approaches for preventing and detecting worms which aim to provide a secure network. In addition, we discussed the attack detection technologies, namely: Intrusion Detection Systems (IDSs), firewalls, and honeypots. A firewall is considered a first line of defense in protecting private information whereas the honeypot is a tool designed to gather the attacker activities; specifically the tools, tactics and, motives of attackers. Three types of honeypots have been mentioned which are Low-interaction honeypots, Mid-interaction honeypots, and High-interaction honeypots. Honeynet has been mentioned as an example of High-interaction honeypots. To reduce the cost of applying the honeynet, there is a solution available now-a-days which is termed "Virtual honeynets" that provides an environment to gather multiple devices operating with different operating systems (OSs) on a single machine. In this work, virtual honeynet technique is used, which will be discussed later. We discussed the related works, and mentioned the features of our system that make it overcome the current automated signature generation systems for polymorphic worms.

Chapter 4 Double-Honeynet: System Theory, and Results

This chapter contains two parts. The first part discusses the design of Double-honeynet system in detail. The second part discusses the followings:

- Brief introduction about the software used to implement the Double-honeynet system.
- Analysis of the outcomes and results.

4.1 Motivation of Double-honeynet System

Unknown Internet worms pose a major threat to Internet infrastructure security, and their destruction causes loss of millions of dollars. Security experts manually generate the IDS signatures by studying the network traces after a new worm has been released. Unfortunately, this job takes a lot of time. We propose a Double-honeynet system that could automatically detect unknown worms without any human interaction. In our system, interaction between the two honeynets works by forming a loop which allows us to collect all polymorphic worm instances which enables the system to produce accurate worm signatures. The Double-honeynet system is a hybrid system with both Network-based and Host-based mechanisms. This allows us to collect polymorphic worm instances at the network-level and host-level, which reduces the false positives and false negatives dramatically.

4.2 Double-Honeynet Architecture

The purpose of our Double-honeynet system is to detect unknown (i.e., previously unreported) worms automatically. A key contribution of this system is the ability of distinguishing worm activities from normal activities without any involvement of experts in the field.

Figure 1 shows the main components of the Double-honeynet system. Firstly, the incoming traffic goes through the Local Router which samples the unwanted inbound connections and redirects the samples connections to Honeynet 1. As the redirected packets pass through the

Local Router, Packet Capture (PCAP) library is used to capture the packets and then to analyze their payloads to contribute to the signature generation process.

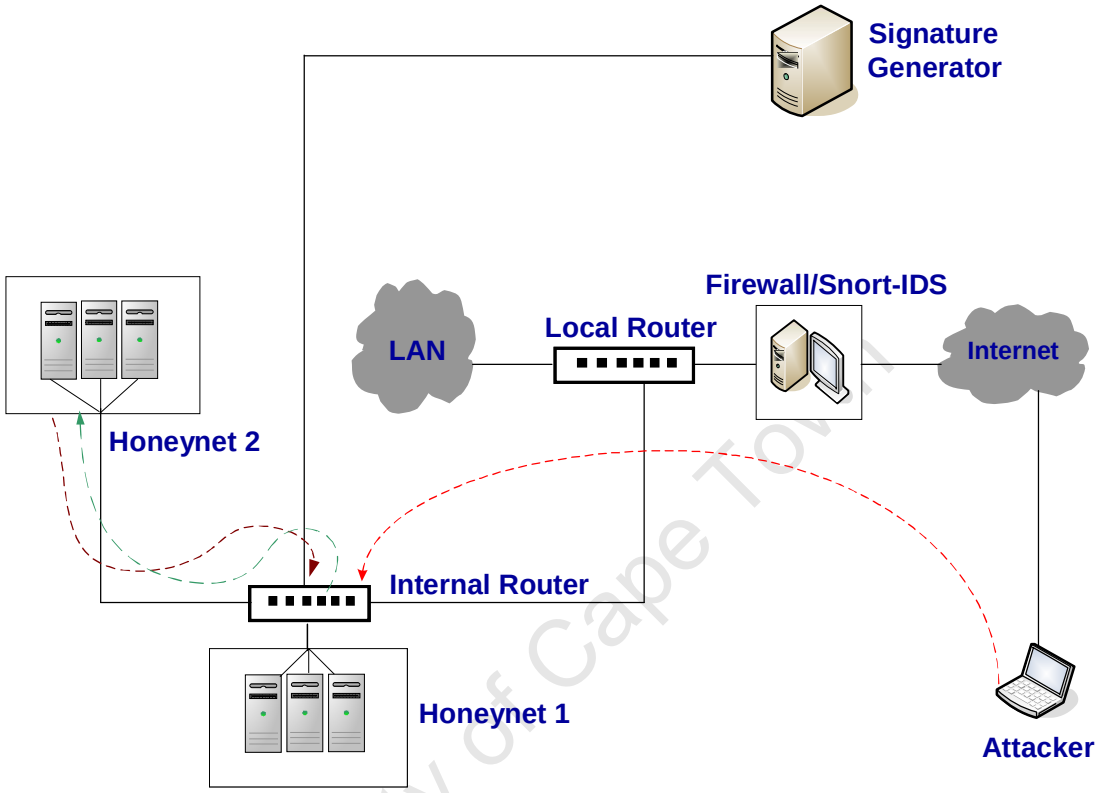

Figure 1. Double-Honeynet System

The Local Router is configured with publicly-accessible addresses, which represent wanted services. Connections made to other addresses are considered unwanted and redirected to Honeynet 1 through the Internal Router. Once Honeynet 1 is compromised; the worm will attempt to make outbound connections to attack another network. The Internal Router is implemented to separate the Double-honeynet from the Local Area Network (LAN). This Router intercepts all outbound connections from Honeynet 1 and redirects those to Honeynet 2, which does the same task forming a loop [61]. The looping mechanism allows us to capture different instances of the polymorphic worm as it mutates on each loop-iteration.

We stop the loop after a considerable amount of time in order to collect polymorphic worms. More details about how much time is taken to collect such types of attacks are presented

in the Section (4.4).

Only those packets that make outbound connections are considered as polymorphic worms, and hence the Double-honeynet system forwards only the packets that make outbound connections. This policy is in place due to the fact that benign users do not try to make outbound connections if they are faced with non-existing addresses. In fact, our system collects other malicious activities which do not intend to propagate themselves but to attack targeted machines only. Such malicious attack is out of our work scope.

When enough instances of worm payloads are collected by Honeynet 1 and Honeynet 2, they are forwarded to the Signature Generator component which generates signatures automatically using specific algorithms. These algorithms will be discussed in Chapter 5.

For example, as shown in Figure 1, if the Local Router suspects Packet 1 (P_1), Packet 2 (P_2), and Packet 3 (P_3) to be malicious, it redirects them to the Honeynet 1 through the Internal Router. Among these three packets, P_1 and P_2 make outbound connections and Internal Router redirects these outbound connections to Honeynet 2. In Honeynet 2, P_1 and P_2 change their payloads and become $P1'$ and $P2'$ respectively (i.e., $P1'$ and $P2'$ are the instances of P_1 and P_2). Therefore, in this case, $P1'$ and $P2'$ make outbound connections and the Internal Router redirects these connections to Honeynet 1. In Honeynet 1, $P1'$ and $P2'$ change their payloads and become $P1''$ and $P2''$ respectively (i.e., $P1''$ and $P2''$ are also other instances of P_1 and P_2).

Now, P_1 and P_2 are found malicious because of the outbound connections. Therefore, Honeynet 1 forwards P1, $P1''$, P2, $P2''$ to the Signature Generator for signature generation process. Similarly, Honeynet 2 forwards $P1'$ and $P2'$ to the Signature Generator for signature generation process.

In this scenario, P_3 does not make any outbound connection when it gets to Honeynet 1. Therefore, P3 is not considered malicious [20].

4.3 Software

The software tools used in the Double-honeynet system are briefly introduced below.

4.3.1 Honeywall Roo CDROM

The honeywall Roo CDROM version 1.4 is downloaded from the Honeynet Project and Research Alliance. It provides data capture, control and analysis capabilities [51, 52]. Most importantly, it monitors all traffic that go in and out of the honeynet. Honeywall Roo CDROM runs Snort-inline, an Intrusion Prevention System based on the Intrusion Detection System Snort. Snort-inline either drops unwanted packets or modifies them to make them harmless. It records information of all the activities in the honeynet using Sebek. It runs the Sebek server, while the Sebek clients run on the honeypots. The clients then send all captured information to the server. For management and data analysis, it uses the Walleye Web interface. Walleye also works as a maintenance interface, but there is a command line tool and a dialog menu that can also be used to configure and maintain the honeywal.

4.3.2 Sebek

Sebek is a data capture tool which mainly records keystrokes, but also all other types of sys_read data [49]. It records and copies all activity on the machine including changes to files, network communications, etc. The main method it uses is to capture network traffic and reassemble the TCP flow. This is in the case of unencrypted data. Encrypted data is another problem, because Sebek can only reassemble it in its encrypted form. Instead of breaking the encryption, Sebek circumvents it by getting the data from the Operating System's kernel. Sebek has a client-server architecture. On the client side, it resides entirely in the Operating System kernel. Whenever a system call is made, Sebek hijacks it by redirecting it to its own *read()* call. This way Sebek can capture the data prior to encryption and after decryption.

After capturing the data, the client sends it to the server, which saves it in a database or simply logs the records. The server is normally on the honeywall machine in the case of a honeynet, and it collects data from all the honeypots and puts it all together for analysis.

To prevent detection by intruders, Sebek employs some obfuscation methods. On the client, it is completely hidden from the user and therefore, from an intruder on the system as well.

This is however not enough because the data captured has to be sent to the Server, thereby exposing itself. Sebek uses a covert channel to communicate with the server. It generates packets to be sent inside Sebek without using the TCP/IP stack and the packets are sent directly to the driver bypassing the raw socket interface. The packets are then invisible to the user and Sebek modifies the kernel to prevent the user from blocking transition of the packets. Figure 2 shows Sebek Deployment.

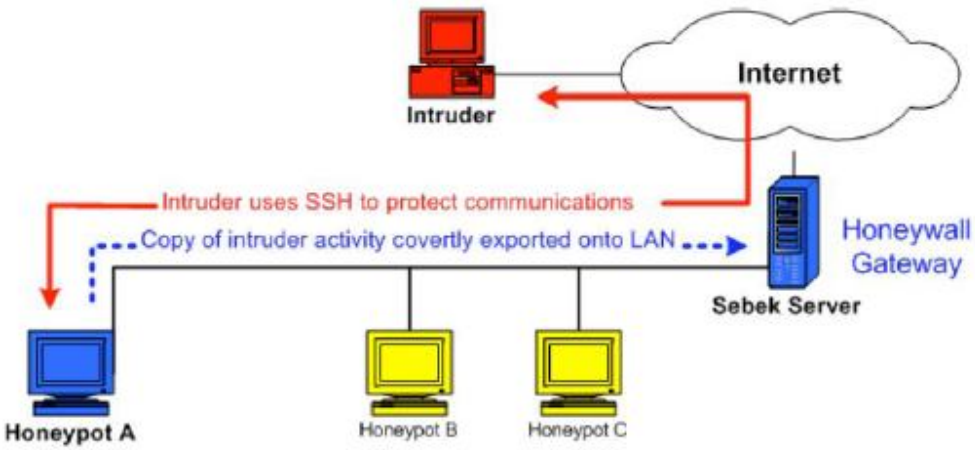

Figure 2. Sebek Deployment

In the case of multiple clients, there is a risk of the clients seeing each other's packets. Sebek configures its own raw socket interface on the clients to ignore all incoming Sebek packets. Only the server can receive Sebek packets. Due to its comprehensive log capabilities, it can be used as a tool for forensics data collection. It has a Web interface that can perform data analysis.

4.3.3 Snort-inline

Snort_inline is a modified version of Snort. It is "an Intrusion Prevention System (IPS) that uses existing Intrusion Detection System (IDS) signatures to make decisions on packets that traverse snort_inline". The decisions are usually drop, reject, modify, or allow [53]. The next section discusses results and analysis of the Double-honeyent system. Double-honeynet system configurations will be discussed in Appendix B.

4.4 Double-honeynet Results

After the successful implementation of Double-honeynet network, we connected the Double-honeynet network to the Internet for seven days to collect attacks. Of course, we checked it periodically to make sure that the network is working properly, to see if the Honeywall serves its purposes, and to examine the data collected. During the seven days, our Double-honeynet system has collected different polymorphic worms instances and other attacks data. We used Walleye, the Honeywall's Web graphical user interface, to analyze data.

To simulate the detection of unknown polymorphic worms, we removed the current signature patterns for the Allaple, Conficker, Balster, and Sasser polymorphic worms from all the Snort instances so that our system can generate new signatures as if the worm is unknown (i.e., not previously recorded).

Once our system has run with the activated polymorphic worm, a new signature would be generated that can be exported to Snort or Bro IDS.

Table 1 shows some samples of polymorphic worms that were captured by the Double-honeynet system.

Table 1. Polymorphic Worm Instances

Polymorphic worm	Number of instances
Allaple worm	3511
Conficker worm	3228
Blaster worm	2817
Sasser worm	2452

4.5 Chapter Summary

This chapter discussed two parts. In the first part, we gave full details about the Double-honeynet system components. In the second part, we gave a brief introduction about the software used to implement the Double-honeynet system and showed the results that were collected for the instances of polymorphic worms.

Chapter 5 Signature Generation Algorithms and Experimental Implementation Results

This chapter discusses two parts. The first part presents our proposed Substring Exaction Algorithm (SEA), Modified Knuth–Morris–Pratt (MKMP) algorithm, and Modified Principal Component Analysis (MPCA), which are used to generate worm signatures from a collection of worm variants captured by our Double-honeynet system. Pseudo codes for these algorithms will be discussed in Appendix A.

To explain how our proposed algorithms generate signatures for polymorphic worms, we assume that we have a polymorphic worm A, that has n instances (A1, A2,.., An). Generating a signature for polymorphic worm A involves two steps:

- First, we generate the signature itself.
- Second, we test the quality of the generated signature by using a mixed traffic (new variants of polymorphic worm A, and normal traffic).

Before stating the details of our contributions and the subsequent analysis part, we briefly mention an introduction about string matching search method and the original Knuth–Morris–Pratt algorithm to give a clear picture of the subject topic.

The second part discusses the implementation results of our proposed algorithms.

5.1 An Overview and Motivation for Using String Matching

After presenting the Double-honeynet system and its functions in the previous Chapter, in this and the following sections, we will describe Substring Exaction Algorithm (SEA), Modified MKMP algorithm, and Modified PCA to highlight our contributions.

String matching [5] is an important subject in the wider domain of text processing. String matching algorithms are basic components used in implementations of practical software used in most of the available operating systems. Moreover, they emphasize programming methods that

serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems.

String matching generally consists of finding a substring (called a pattern) within another string (called the text). The pattern is generally denoted as,

$$x = x[0..m\text{-}1]$$

Whose length is m and the text is generally denoted as

$$y = y[0..n\text{-}1]$$

Whose length is n. Both the strings-pattern and text are built over a finite set of characters which is called the alphabet and denoted by Σ whose size is denoted by σ.

The string matching algorithm plays an important role in network intrusion detection systems (IDS), which can detect malicious attacks and protect the network systems. In fact, at the heart of almost every modern intrusion detection system, there is a string matching algorithm. This is a very crucial technique because it allows detection systems to base their actions on the content that is actually flowing to a machine. From a vast number of packets, the string identifies those packets that contain data, matching the fingerprint of a known attack. Essentially, the string matching algorithm compares the set of strings in the rule-set with the data seen in the packets, which flow across the network.

Our work uses Substring Extraction Algorithm (SEA) and Modified Knuth–Morris–Pratt (MKMP) algorithm (which are based on string matching algorithms), to generate signatures for polymorphic worm attacks. The SEA aims at extracting substrings from polymorphic worm, whereas MKMP algorithm aims to find out multiple invariant substrings that are shared between polymorphic worm instances and to use them as signatures.

5.2 The Knuth–Morris–Pratt Algorithm (KMP)

The Knuth–Morris–Pratt string searching algorithm (or, KMP algorithm) [5] searches for occurrences of a "word", *W*, within a main "text string", *S*, by employing the observation that

when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters [5, 43].

Let us take an example to illustrate how the algorithm works. To illustrate the algorithm's working method, we will go through a sample run (relatively artificial) of the algorithm. At any given time, the algorithm is in a state determined by two integers, *m* and *i*. *m* denotes the position within *S* which is the beginning of a prospective match for *W*, and *i* denotes the index in *W* denoting the character currently under consideration. This is depicted at the start of the run, like:

```
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

We proceed by comparing successive characters of W to "parallel" positional characters of *S*, moving from one to the next if they match. However, in the fourth step in our noted case, we get that *S[3]* is a space and *W[3]* is equal to the character D (i.e., *W[3]* = 'D'), which is a mismatch. Rather than beginning to search again at the position S [1], we note that no 'A' occurs between positions 0 and 3 in S except at 0. Hence, having checked all those characters previously, we know that there is no chance of finding the beginning of a match if we check them again. Therefore, we simply move on to the next character, setting *m = 4* and *i = 0*.

```
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456
```

We quickly obtain a nearly complete match "ABCDAB", but when at W[6] (S[10]), we again have a discrepancy. However, just prior to the end of the current partial match, we passed an "AB" which could be the beginning of a new match, so we must take this into consideration. As we already know that these characters match the two characters prior to the current position, we need not check them again; we simply reset *m = 8*, *i = 2*, and continue matching the current character. Thus, not only do we omit previously matched characters of *S* but also previously matched characters of *W*.

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

We continue with the same method of matching, till we match the word *W*.

5.2.1 Proposed Substrings Extraction Algorithm (SEA)

In this subsection, we show how our proposed Substring Extraction Algorithm (SEA) is used to extract substrings form one of the polymorphic worm variants that are collected by the Double-honeynet system.

This subsection and the next one (5.2.2) show the signature generation process for polymorphic worm A using the SEA and a Modified Knuth–Morris–Pratt Algorithm (MKMP algorithm). The procedure of testing the quality of the generated signature will be discussed in Subsection (5.2.3)

Let us assume that we have a polymorphic worm *A*, that has *n* instances (A_1,..., A_n) and A_i has length M_i for i=1,…,n. Assume that A_1 selected to be the instance from which we extract substrings and the A1 string contains a_1 a_2 a_3... a_{m1}. Let, X to be the minimum length of a substring that we are going to extract from A_1. The first substring from A_1 with length X, is (a_1 a_2 ... a_x). Then, we shift one position to the right to extract a new substring, which will be (a_2 a_3... a_{x+1}). Continuing this way, the last substring from A_1 will be (a_{m1-X+1}... a_{m1}). In general, if instance A_i has length equal to M, and let a minimum length of the substring that we are going to extract from A_1 equals to X, then the Total Number of Substrings (TNS) that will be extracted from A_i could be obtained by this equation:

$$TNS\ (A_i) = M\text{-}X+1$$

The next step is to increase X by one and start new substrings extraction from the beginning of A_1. The first substring will be (a_1 a_2 ... a_{x+1}). The substrings extraction will continue satisfying this condition, X< M.

Figure 3 and Table 2 show all substrings extraction possibilities using the proposed Substring Extraction Algorithm (SEA) from the string ZYXCBA assuming the minimum length of X is equal to three.

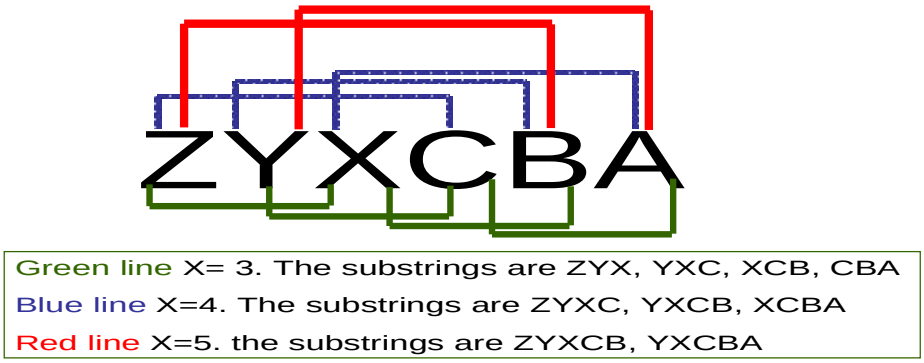

Figure 3. Extraction Substrings

Table 2. Substrings Extraction

No. of Subtractions	Length of X	Substrings
S1,1	3	ZYX
S1,2	3	YXC
S1,3	3	XCB
S1,4	3	CBA
S1,5	4	ZYXC
S1,6	4	YXCB
S1,7	4	XCBA
S1,8	5	ZYXCB
S1,9	5	YXCBA

The output of the SEA will be used by both the Modified Knuth–Morris–Pratt Algorithm (MKMPA) and the Modified PCA (MPCA) method. The MKMPA uses the substrings extracted

by the SEA to search the occurrences of each substring in the remaining of the instances (A_2, A_3,…,A_n). The substrings that occur in all the remaining instances will be considered as worm signature. To clarify some of the points noted here, we will present the details of MKMPA in the next subsection.

5.2.2 A Modified Knuth–Morris–Pratt Algorithm (MKMP algorithm)

In this subsection, we describe our modification on Knuth-Morris-Pratt Algorithm. As we mentioned in Section (5.2), the Knuth-Morris-Pratt algorithm searches for occurrences of W (word) within S (text string). Our modification on the KMP algorithm is to search for occurrence of different words (W_1, W_2…, W_n) within string text "S". For example, say we have a polymorphic worm, A with N instances (A_1, A_2,..., A_n). Let us select A_1 to be the instance from which we would extract substrings. If 9 substrings are extracted from A_1, each substring will be W_i for i= 1 to 9. That means, A_1 has 9 Words (W_1, W_2…, W_9) whereas the remaining instances (A_2, A_3,..., A_n) are considered as S "text string" [65].

Considering the above example, the Modified KMP algorithm (MKMP) algorithm searches the occurrences of W_1 in the remaining instances of S (A_2, A_3,..., A_n). If W_1 occurs in all remaining instances of S, then we consider it as signature otherwise we ignore it. The other words (W_2 , W_3, ,…, W_9) are similarly dealt with. Just as an example, if W_1, W_5 ,W_6, and W_9 occur in all remaining instances of S, then W_1, W_5 ,W_6, and W_9 are considered a signature of the polymorphic worm A.

5.2.3 Testing the quality of the generated signature for Polymorphic Worm A

We test the quality of the generated signature for polymorphic worm A by using a mixed traffic (new variants of polymorphic worm A & normal traffic, i.e. innocuous packets). The new variants of polymorphic worm A are not the same variants that are used to generate the signature. Let us assume that our system received a packet P (where P contains either malicious or innocuous data). The MKMP algorithm compares P payload against the generated signature to determine whether P is a new variant of polymorphic worm A or not. The MKMP algorithm

considers P as a new variant of the polymorphic worm A, if all the substrings of the generated signature appear in P.

5.3 A Modified Principal Component Analysis (MPCA)

Before introducing the MPCA, we give a brief introduction and motivation of using PCA statistical method in our work. Then we will illustrate the MPCA which contains our contributions in the PCA.

5.3.1 An Overview and Motivation of Using PCA in Our Work

In general, when presented with the need to analyze a high dimensional structure, a commonly-employed and powerful approach is to seek an alternative lower-dimensional approximation to the structure that preserves its important properties. A structure that can often appear complex because of its high dimension may be largely governed by a small set of independent variables and so can be well approximated by a lower dimensional representation. Dimension analysis and dimension reduction techniques attempt to find these simple variables and can therefore be a useful tool to understand the original structures. The most commonly used technique to analyze high dimensional structures is the method of Principal Component Analysis [46]. Given a high dimensional object and its associated coordinate space, PCA finds a new coordinate space which is the best one to use for dimension reduction of the given object. Once the object is placed into this new coordinate space, projecting the object onto a subset of the axes can be done in a way that minimizes error. When a high-dimensional object can be well approximated in this way in a smaller number of dimensions, we refer to the smaller number of dimensions as the object's intrinsic dimensionality.

5.3.2 Our Contributions in the PCA

This subsection, Subsection (5.3.3), and Subsection (5.3.4) show the signature generation process for polymorphic worm A using a Modified Principle Component Analysis (MPCA). Testing the quality of the generated signature will be discussed in Subsection (5.3.5).

In our work, instead of applying PCA directly, we have made appropriate modifications to

it to fit it with our mechanism. Our contribution in the PCA method is in combining the PCA (i.e., extend) with the proposed Substring Extraction Algorithm (SEA) to get more accurate and relatively faster signatures for polymorphic worms. The extended method (SEA & PCA) is termed Modified Principle Component Analysis (MPCA). We have previously mentioned that the polymorphic worm evades the IDSs by changing its payload in every infection attempts; however, there are some invariant substrings that will remain fixed (i.e., some substrings will not change) in all polymorphic worm variants, so the Substring Extraction Algorithm (SEA) extracts substrings from polymorphic worm by a good way (i.e., it will extract all the possibilities of substrings from a polymorphic worm variant, which contain worm signature) that helps us to get accurate signatures. After the SEA extracts the substrings, it will pass those to the PCA, thus easing the heavy burden to the PCA in terms of time (i.e., the PCA directly will start by determining the Frequency Count of each substring in rest of the instances without doing substring extraction process).

After the PCA receives the substrings from SEA, it will determine the Frequency Count of each substring in the remaining instances (A_2, A_3…, A_n). Lastly, the PCA will determine the most significant data on the polymorphic worm instances and use them as signature [50]. We present the details in the next subsection.

5.3.3 Determination of Frequency Counts

Here, we determine the frequency count of each substring S_i (A_1 substrings), in each of the remaining instances (A_2 ,..., A_n). Then, we apply the Principal Component Analysis (PCA) on the frequency count data to reduce the dimension and get the most significant data.

5.3.4 Using PCA to Determine the Most Significant Data on Polymorphic Worm Instances

The methodology of employing PCA to the given problem is outlined below.

Let F_i denotes the vector of frequencies (F_{i1} ,..., F_{iN}) of the substring S_i in the instances (A_1,..., A_n), i= 1,...,L.

We construct the frequency matrix F by letting F_i be the i^{th} row of F, provided that F_i is not the zero vector.

$$F=\begin{pmatrix} f_{11} & \Lambda & f_{1N} \\ M & O & M \\ f_{L1} & \Lambda & f_{LN} \end{pmatrix}$$

5.3.4.1 Normalization of data

The normalization of the data is applied by normalizing the data in each row of the matrix F, yielding a matrix D (L x N).

$$D=\begin{pmatrix} d_{11} & \Lambda & d_{1N} \\ M & O & M \\ d_{L1} & \Lambda & d_{LN} \end{pmatrix}$$

$$\text{dik} \leftarrow \frac{f_{ik}}{\sum_{j=1}^{N} f_{ij}}$$

5.3.4.2 Mean adjusted data

To get the data adjusted around zero mean, we use the formula:

$$g_{ik} \leftarrow d_{ik} - \bar{d}_i \quad \forall i,k$$

Where $\bar{d}_i$ = mean of the i^{th} vector

$$= \frac{1}{N}\sum_{j=1}^{N} d_{ij}$$

The data adjust matrix G is given by:

$$G=\begin{pmatrix} g_{11} & \Lambda & g_{1N} \\ M & O & M \\ g_{L1} & \Lambda & g_{LN} \end{pmatrix}$$

5.3.4.3 Evaluation of the covariance matrix

Let g_i denotes the i^{th} row of G, then the covariance between any two vectors g_i and g_j is given by:

$$\text{Cov}(g_i, g_j) = C_{ij} = \frac{\sum_{k=1}^{L}(d_{ik} - \bar{d}_i)(d_{jk} - \bar{d}_j)}{N-1}$$

Then the covariance matrix C (N x N) is given by:

$$C = \begin{pmatrix} C_{11} & \Lambda & C_{1N} \\ M & O & M \\ C_{N1} & \Lambda & C_{NN} \end{pmatrix}$$

5.3.4.4 Eigenvalue Evaluation

Evaluate the eigenvalues of the matrix C from its characteristic polynomial $|C - \lambda I| = 0$, and then, compute the corresponding eigenvectors.

5.3.4.5 Principal Component Evaluation

Let $L_1, L_2, \ldots, L_N$ be the eigenvalues of the matrix C obtained by solving the characteristic equation $|C - \lambda I| = 0$. If necessary resort the eigenvalues of C such in a descending order such that $|L_1| >= \ldots >= |L_N|$. Let $V_1, V_2, \ldots, V_N$ be the eigenvectors of matrix C corresponding to the eigenvalues $L_1, L_2, \ldots, L_N$. The k principal components are given by $V_1, V_2, \ldots, V_K$ where K<=N.

5.3.4.6 Projection of data adjust along the Principal Component

Let V be the matrix which has the k principal components as its columns. That is

$V = [V_1, V_2, \ldots, V_K]$

Then the feature descriptor is obtained from the equation

$$\text{Feature Descriptor} = V^T \text{ x } F.$$

To determine the threshold of polymorphic worm A, we use a distance function (Euclidean distance) to evaluate the maximum distance between the rows of F and the rows of FD. The maximum distance R works as a threshold. The Euclidean distance theory will be discussed in Section (5.4).

5.3.5 Testing the quality of generated signature for Polymorphic Worm A

In the above subsection, we calculated the Feature Descriptor (FD) and threshold for polymorphic worm A. In this subsection, we test the quality of generated signature for polymorphic worm A by using a mixed traffic (new variants of polymorphic worm & normal traffic i.e. innocuous packets), the new variants of polymorphic worm A are not the same variants that are used to generate the signature.

Let us assume that our system received a packet P (where P contains either malicious or innocuous data). The MPCA performs the following steps to determine whether P is a new variant of polymorphic worm A or not:

- Determine frequencies of the substrings of W array in P, (W array contains extracted substrings of A1, as we mentioned earlier). This will produce a frequency matrix F1.

- Calculate the distance between the polymorphic worm FD and F1 using Euclidean distance. This will produce a distance matrix D1.

    - Compare the distances in D1 to the threshold R of polymorphic worm A. If any <= the threshold, classify P as a new variant of polymorphic worm A.

5.4 Clustering Method for Different Types of Polymorphic Worms

When our network receives different types of polymorphic worms (mixed polymorphic worms), we must first separate them into clusters and then generate signatures to each cluster as the same as in section (5.2, and 5.3). To perform the clustering we use Euclidean distance which is the most familiar distance metric. Euclidean distance is frequently used as a measure of

similarity in the nearest neighbor method [46]. Let X=(X_1, X_1,..., X_p)' and Y= (Y_1, Y_2,..., Y_p)' . The Euclidean distance between X and Y is:

$$d(x,y) = \sqrt{(x-y)'(x-y)}$$

5.5 MATLAB and Experiments Done in Our Work

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using the MATLAB product, one can solve technical computing problems faster than that of using traditional programming languages, such as C, C++, and Fortran. We can use MATLAB in a wide range of applications, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, and computational biology. MATLAB provides a number of features for documenting and sharing our work. We can integrate our MATLAB code with other languages and applications, and distribute MATLAB algorithms and applications [16].

We perform experiments to demonstrate the effectiveness of the proposed signature generation algorithms which are Modified Knuth–Morris–Pratt (MKMP algorithm) and Modified Principal Component Analysis (MPCA) in identifying polymorphic worms. The malicious payloads of Conficker worm, Allaple worm, Blaster worm, and Sasser worm are used in the experiments. 200 instances of each of the above worms are used in this experiment. We use 100 instances from each of the worms for signature generation. The rest of the instances are mixed with normal traffic (i.e. innocuous packets) to test the quality of the generated signature of each of the worms. We used Matlab code running on a PC with Intel Pentium 4, 3.19-GHZ CPU and 8.00 GB RAM.

We consider mixed traffic as worm and normal traffic. The percentages of worm and normal traffic of mixed traffic are 20% and 80% respectively.

5.6 Signature Generation process for Polymorphic Worms Using Modified Knuth–Morris–Pratt Algorithm (MKMP Algorithm)

In this section, we show how we generate signatures for polymorphic worms using Modified Knuth–Morris–Pratt Algorithm (MKMP algorithm). The generation of signatures involves two steps: First, we generate the signature itself. Second, we calculate the detection rate, false positives and false negatives to test the quality of the generated signatures.

5.6.1 Signature Generation Process

Here, we describe the signature generation process for Allaple worm, Conficker worm, Blaster worm, and Sasser worm.

Figures 4, 5, 6, and 7 show the experimental results for the above four worms based on Modified Knuth–Morris–Pratt algorithm (MKMP algorithm), which first identifies the invariant substrings that are shared between all worm instances. Second, the invariants substrings are used as signatures to match against the test instances. Based on the figures shown below we find that as the number of sample variants increases, the signature width decreases. A shorter signature increases the chance of appearing in normal traffic. Consequently, the false negative ratio decreases, but the false positive ratio increases dramatically. Again, as in the figures below, the false negatives increase dramatically when the number of the sample variants decrease. In order to get good results for our work in terms of detection rate, low false positives, and low false negatives, we removed the shorter signatures and longer signatures from the generated signatures. After we removed theses signatures (shorter and longer signatures), we got a high detection rate, zero false positives, and low false negatives as shown in section (5.6.2, and 5.6.3).

From our analysis and experiments, we find that 100 samples for each worm are enough to generate accurate signatures. Because, as shown in the figures below, when we reached the last sample (sample number 100) of each worm, we found that the signature width became very short and it would increase the false positives dramatically. So, if we increase the number of samples more than 100, more false positives would appear.

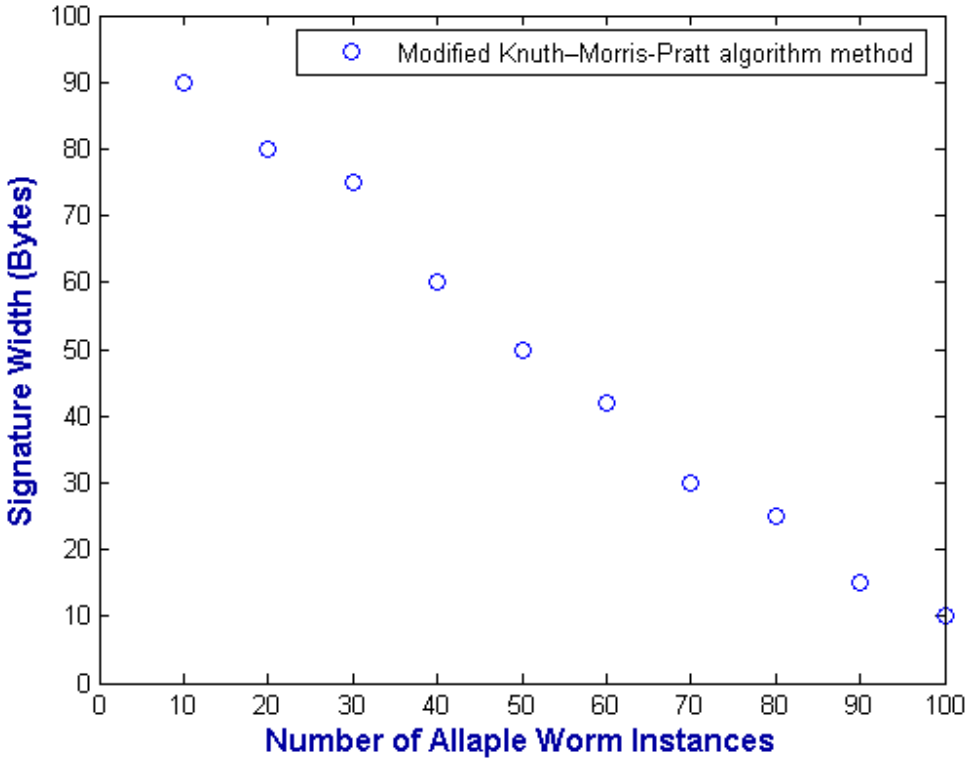

Figure 4. Signature Generation Process for Allaple Worm

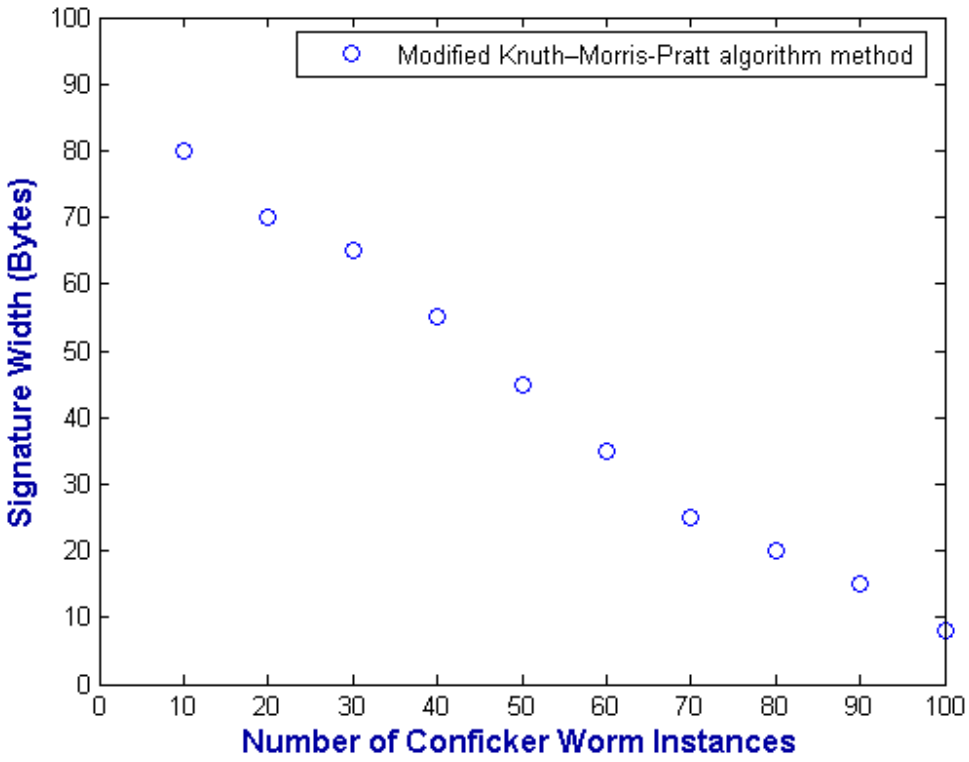

Figure 5. Signature Generation Process for Conficker Worm

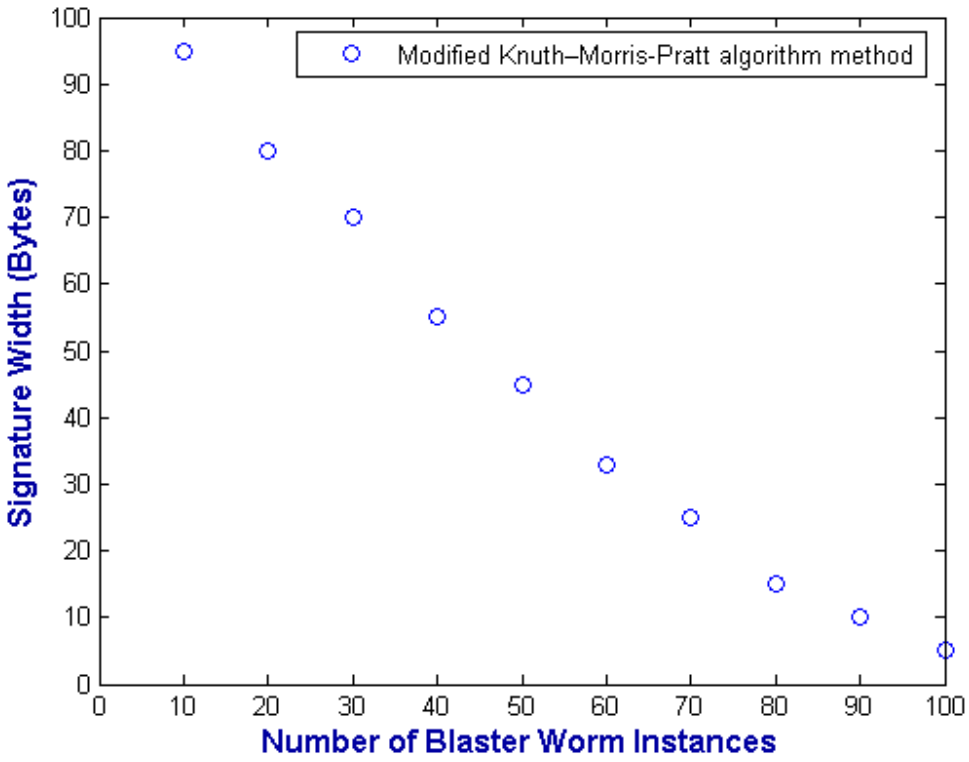

Figure 6. Signature Generation Process for Blaster Worm

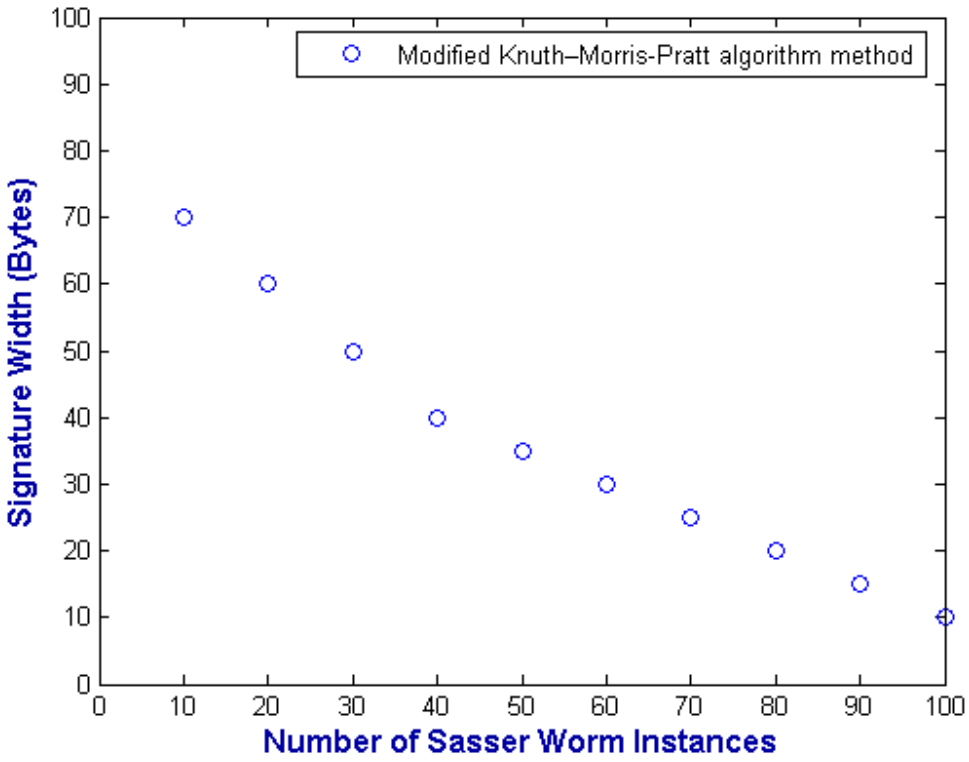

Figure 7. Signature Generation Process for Sasser Worm

5.6.2 MKMP Algorithm Detection Rate

In the following, we test the quality of the generated signatures for the four worms in terms of detection rate.

Figures 8, 9, 10, and 11 show detection rates of the four worms on mixed traffic (worm instances & normal traffic). The X-axis indicates number of worm instances and normal traffic, Y-axis indicates worm detection rate. The detection rate increases as the number of worm instances increases.

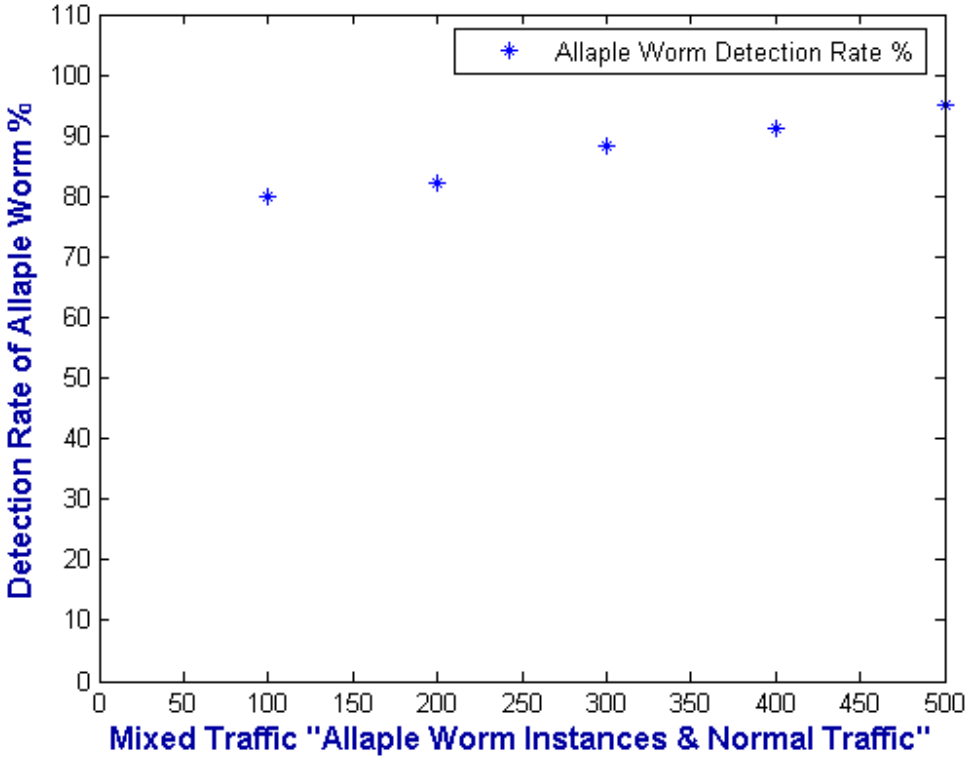

Figure 8. Allaple Worm Detection Rate

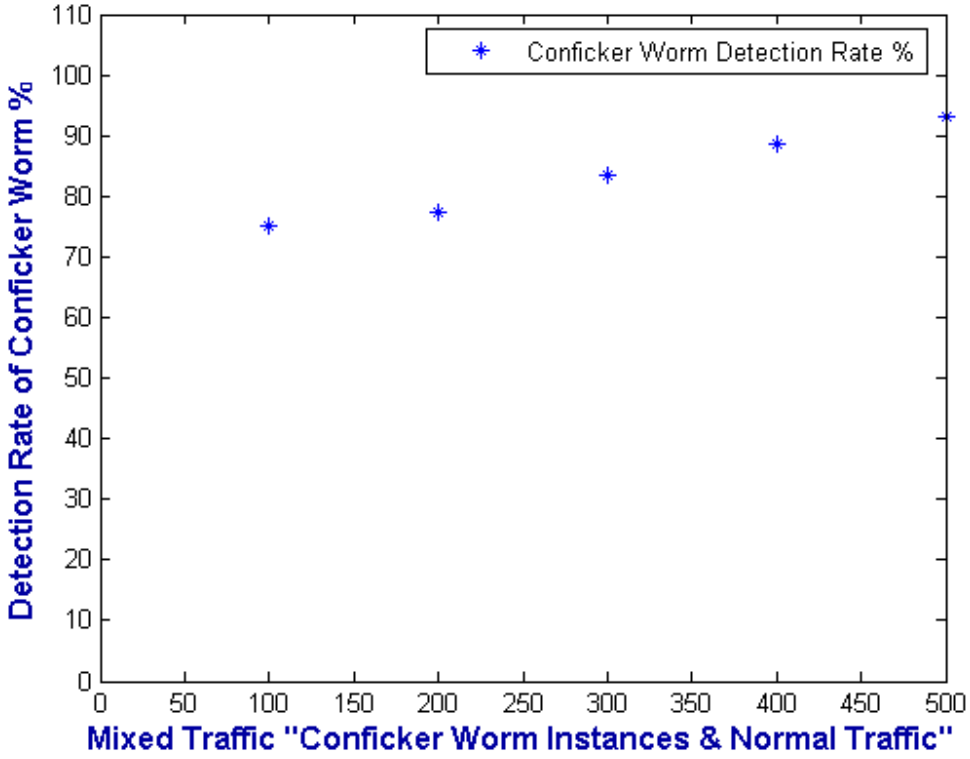

Figure 9. Conficker Worm Detection Rate

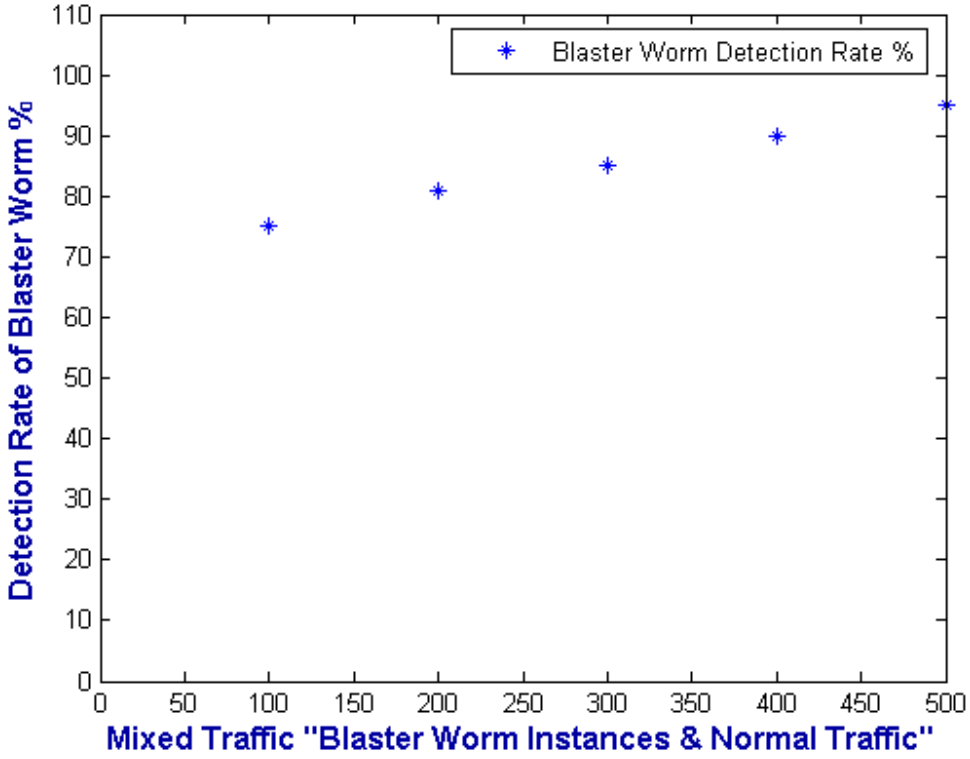

Figure 10. Blaster Worm Detection Rate

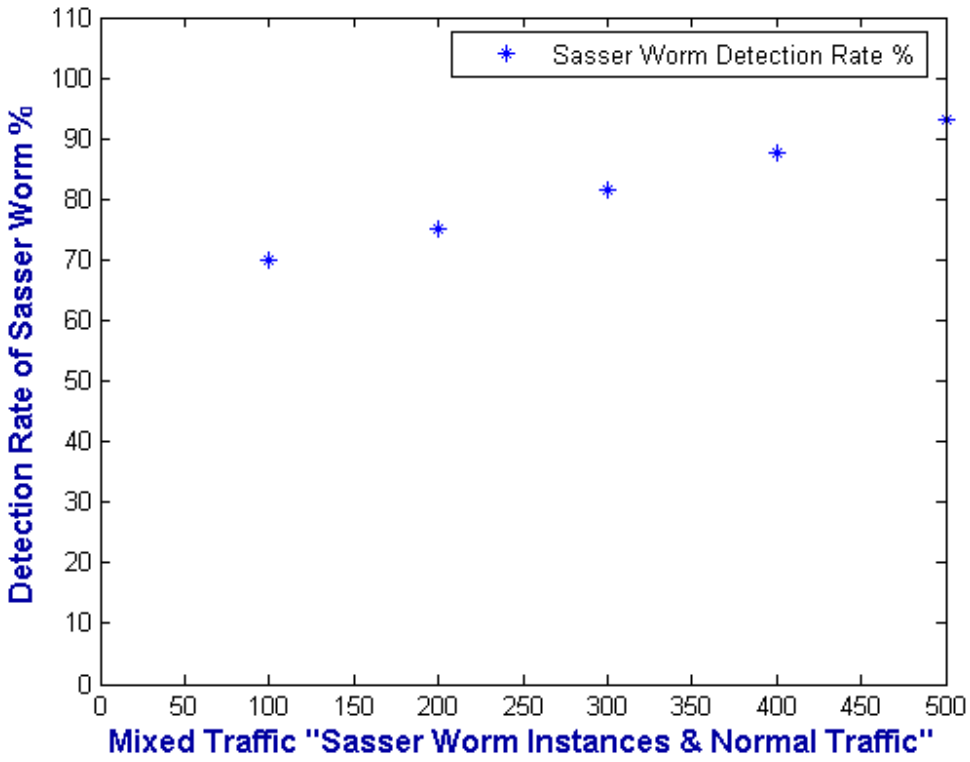

Figure 11. Sasser Worm Detection Rate

5.6.3 False Positives and False Negatives Percentages

In the following, we calculate the percentage of false positives and false negatives for the four worms.

Figures 12, 13, 14, and 15 show the false positive and false negative percentages for the four worms. As shown in the figures, we get zero false positives whereas the false negatives decrease as the number of worm instances increases.

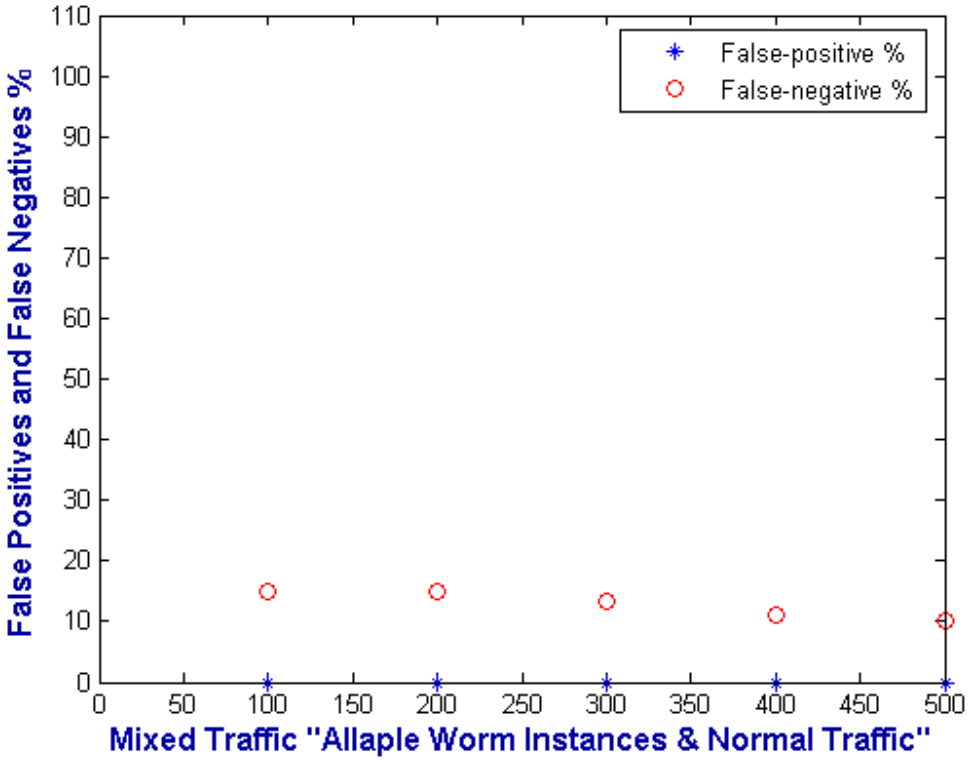

Figure 12. Allaple Worm False Positives and False Negatives

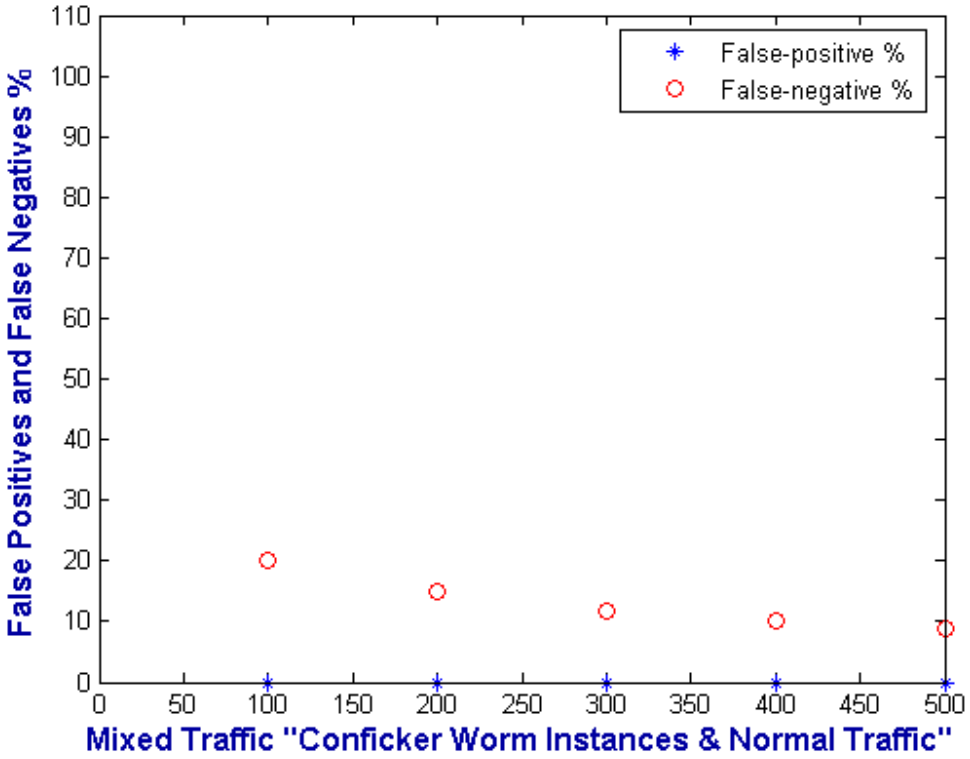

Figure 13. Conficker Worm False Positives and False Negatives

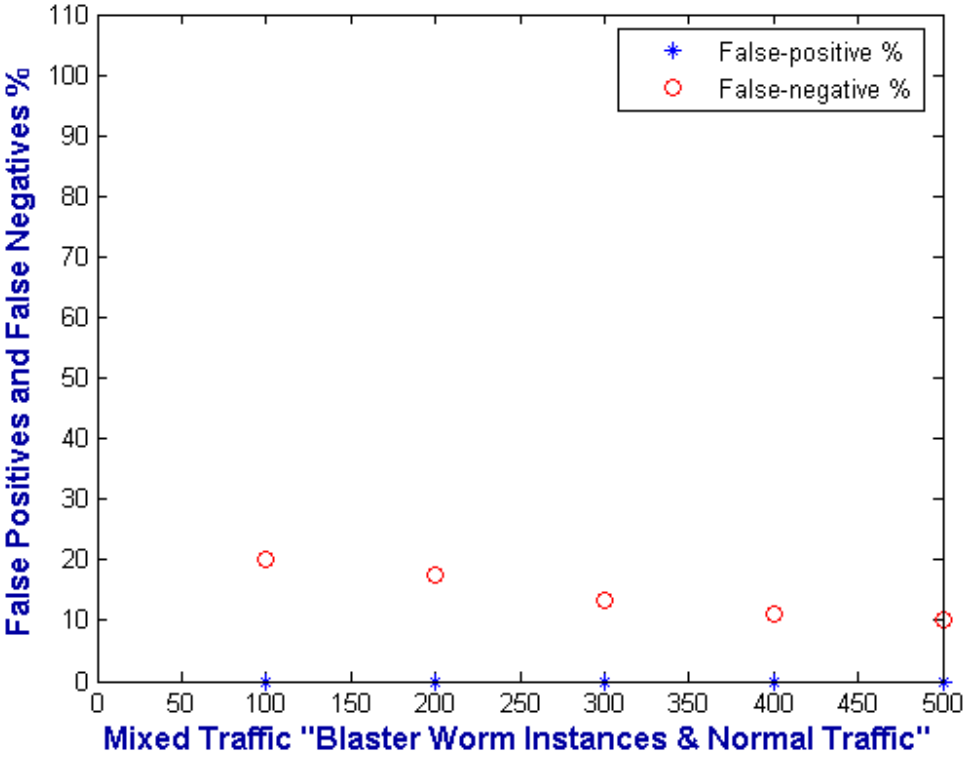

Figure 14. Blaster Worm False Positives and False Negatives

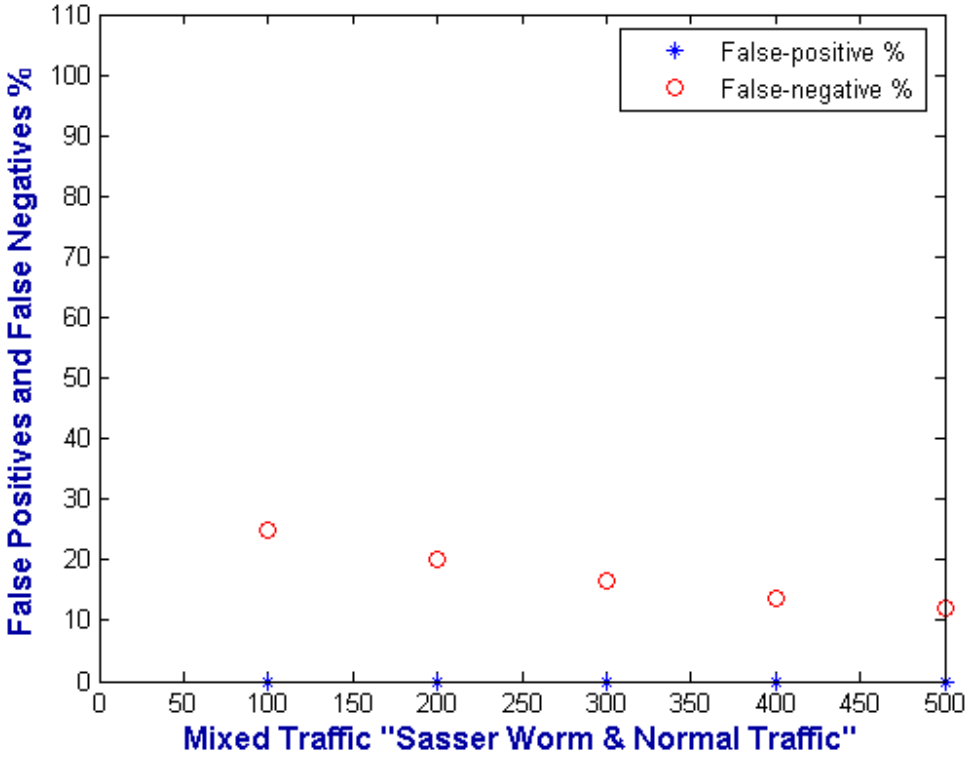

Figure 15. Sasser Worm False Positives and False Negatives

5.7 Signature Generation Process for Polymorphic Worms using Modified Principal Component Analysis (MPCA)

The second method that we use to generate signatures for polymorphic worms is Modified Principal component analysis (MPCA) as we discussed in (5.3).

5.7.1 Signature Generation Process

In the following, we generate signature for Allaple worm, Conficker Worm, Blaster Worm, and Sasser Worm. Then, we calculate detection rate, false positives and false negatives to test the quality of the generated signature.

5.7.1.1 Data before Reduction Process

Now, we describe the worm payload before reduction process for the four worms.

Figures 16, 17, 18, and 19 Show the worm payload before reduction process. The X-axis indicates the number of worm instances that are used to generate signature. The Y-axis indicates frequency count of the extracted substrings. The worm payload dimensions might be high before reduction. Hence, the worm signature might be hard to be found in such dimension.

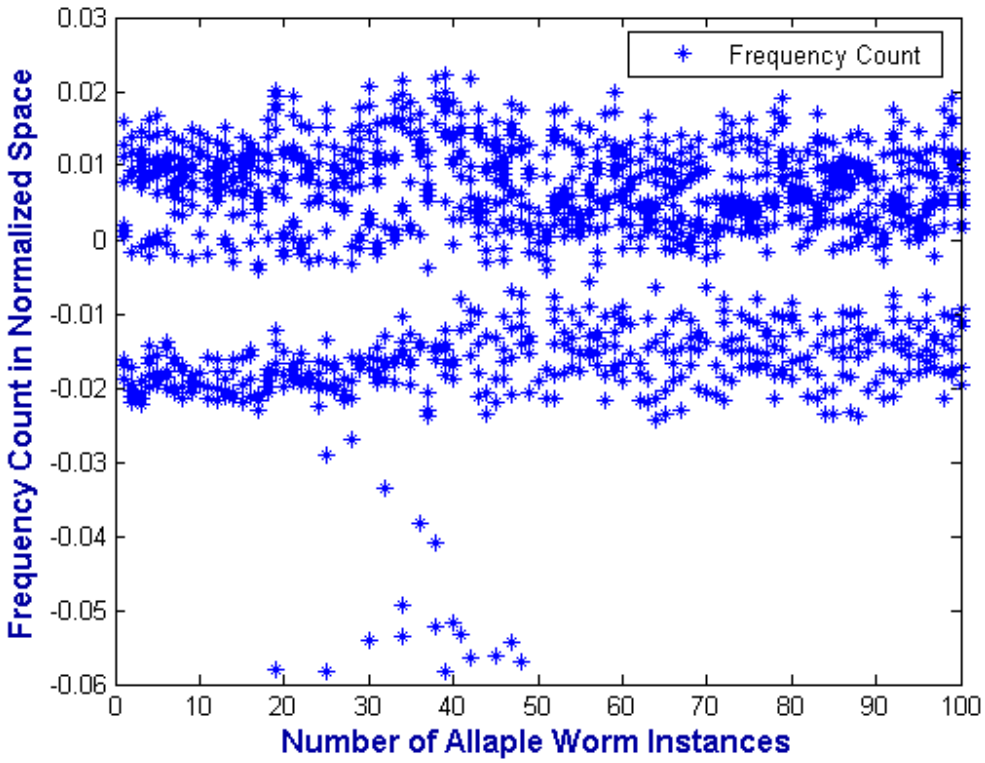

Figure 16. Allaple Worm Substrings before Reduction

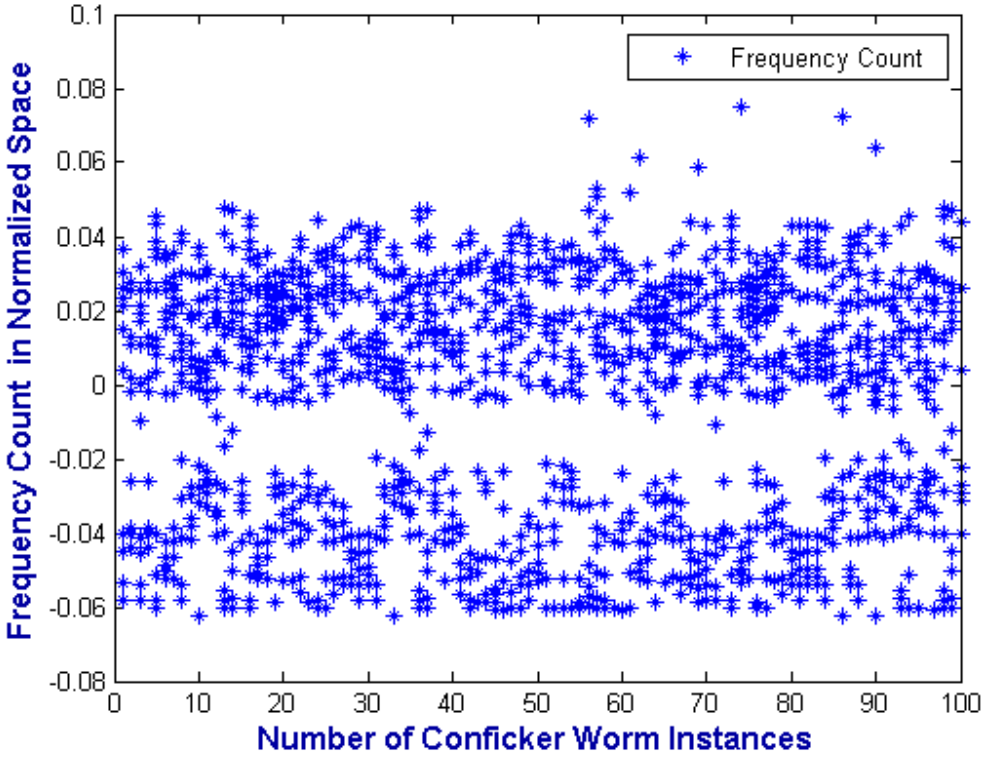

Figure 17. Conficker Worm Substrings before Reduction

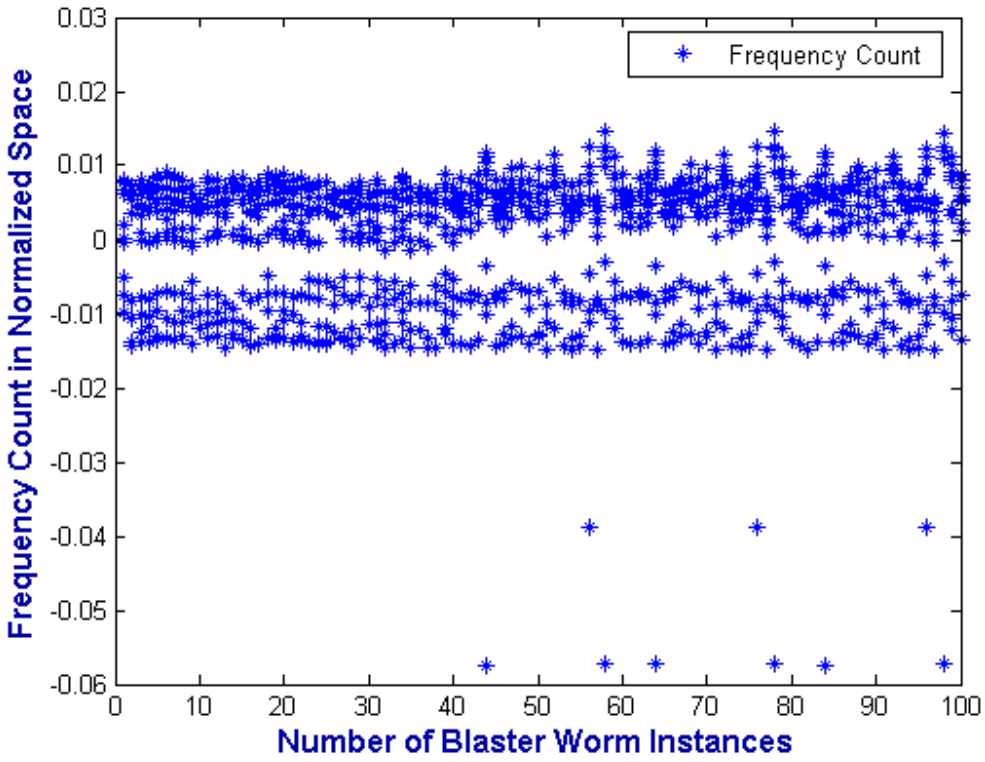

Figure 18. Blaster Worm Substrings before Reduction

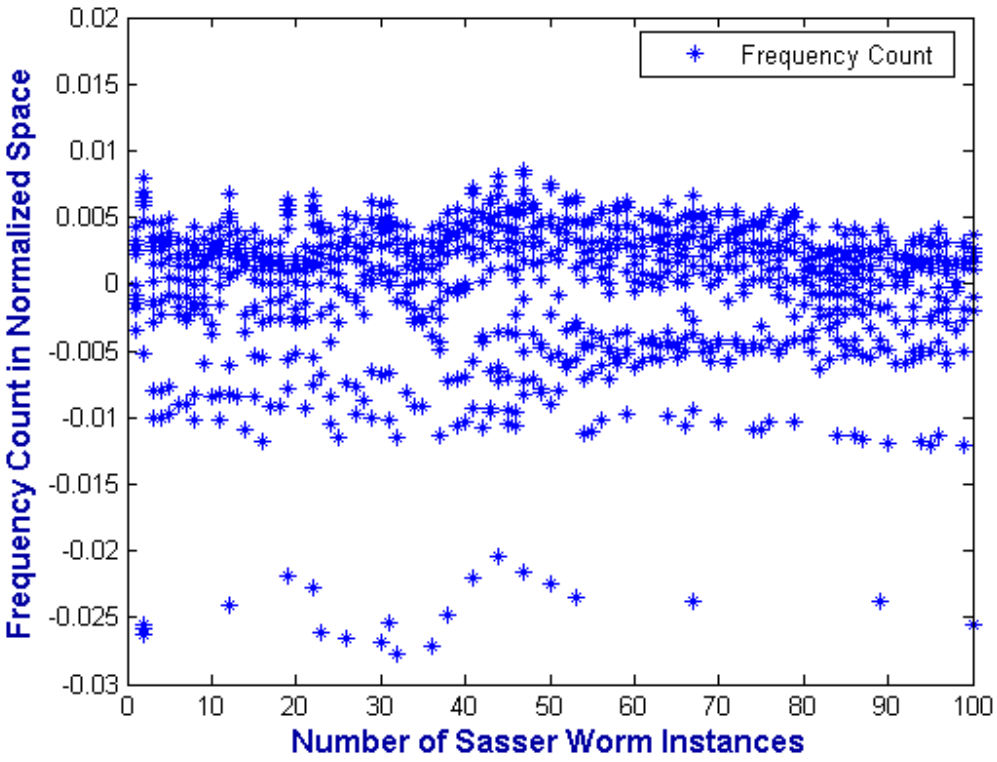

Figure 19. Sasser Worm Substrings before Reduction

5.7.1.2 Data after Reduction Process

In this part, we show how the MPCA can be used to reduce the dimension of the worm payload for the four worms without losing any significant data.

Figures 20, 21, 22, and 23 show the most significant data obtained by the reduction of worm payload using the MPCA. It is noticed that the dimension of worm payload is reduced dramatically.

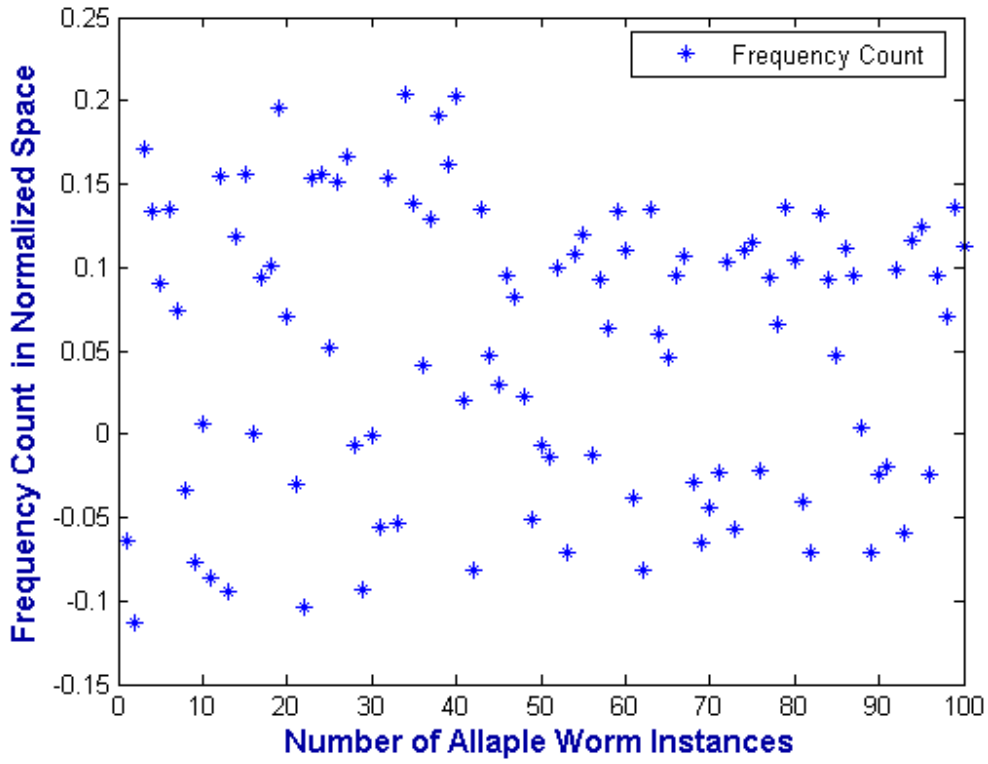

Figure 20. Allaple Worm Substrings after Reduction

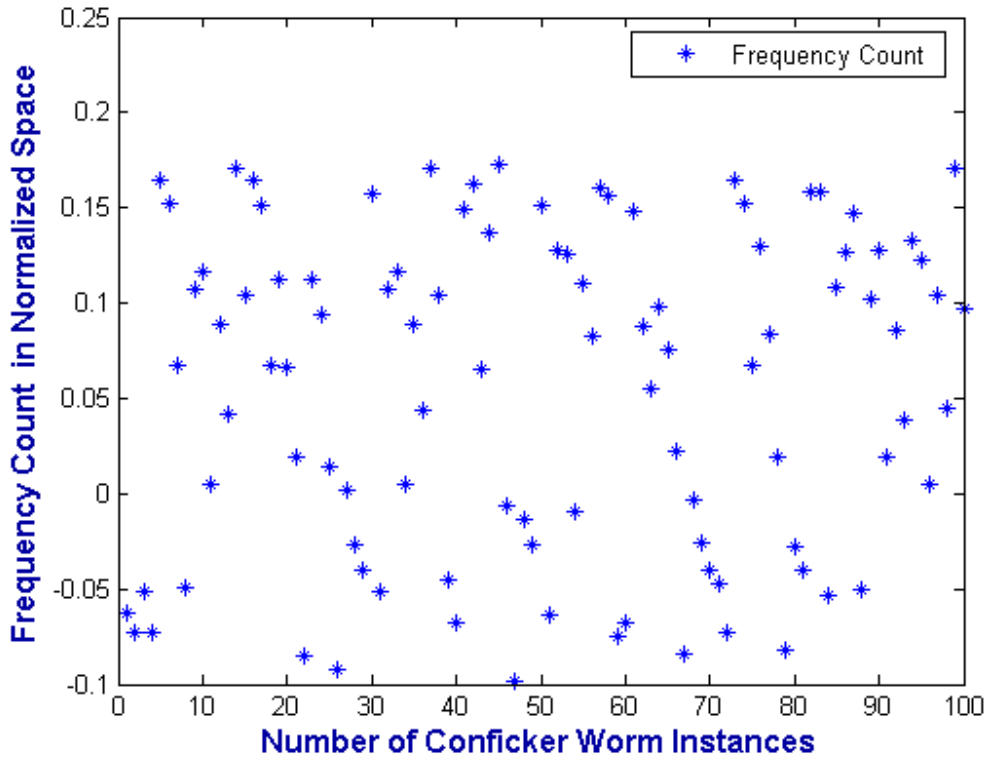

Figure 21. Conficker Worm Substrings after Reduction

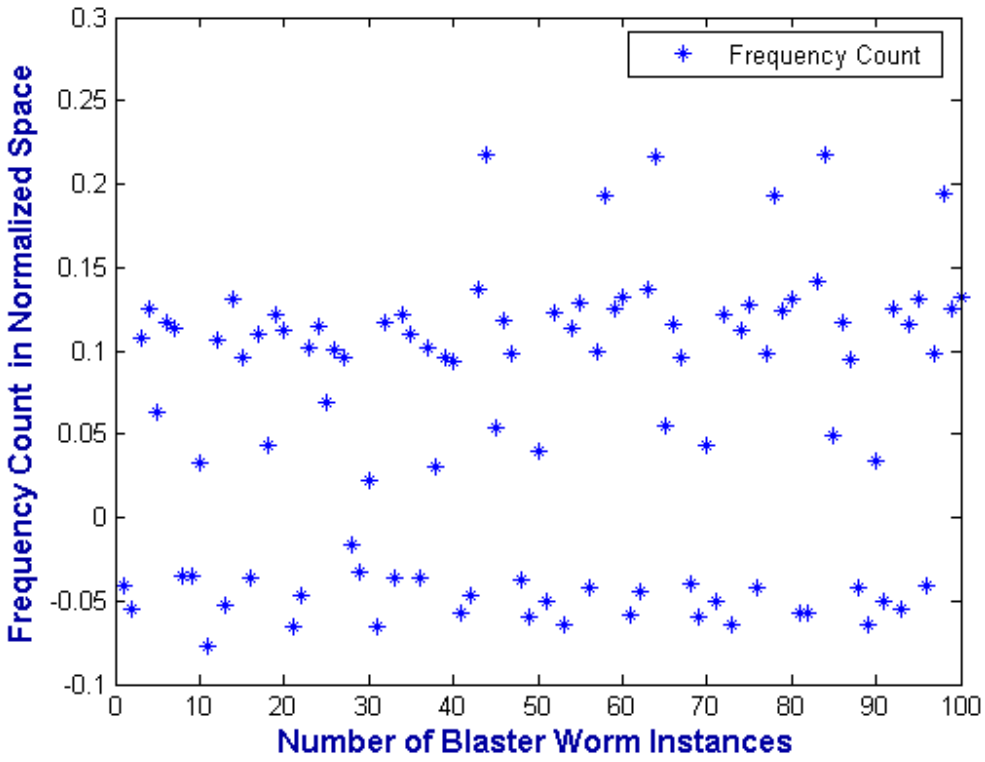

Figure 22. Blaster Worm Substrings after Reduction

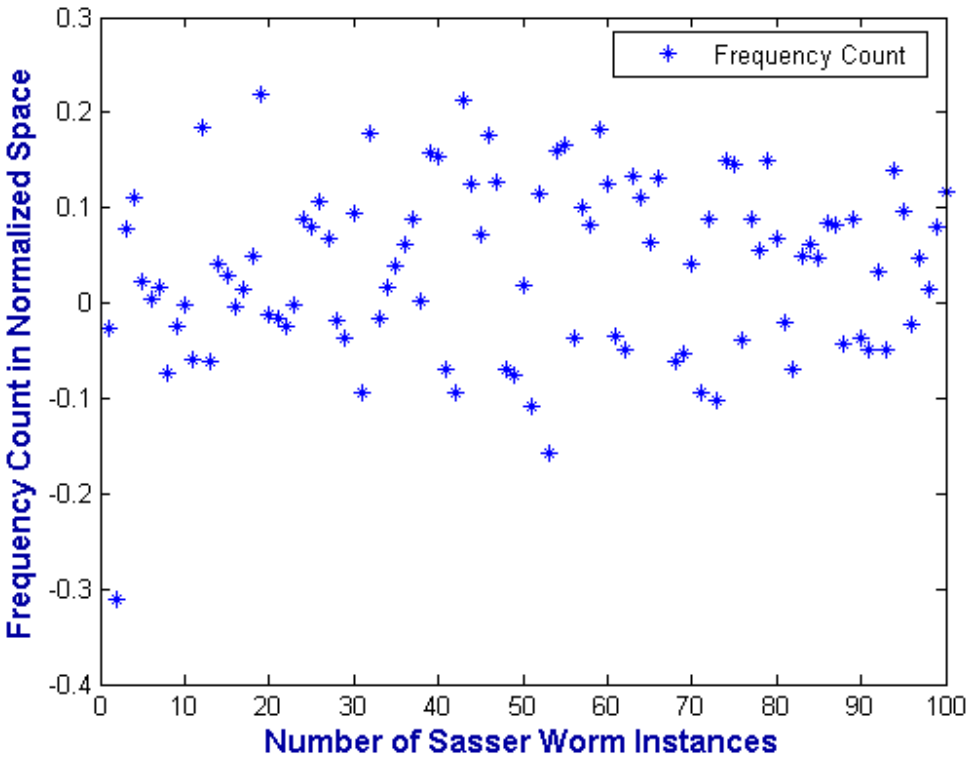

Figure 23. Sasser Worm Substrings after Reduction

5.7.1.3 Detection Rate

Here, we test the quality of the generated signature of the four worms in terms of detection rate.

Figures 24, 25, 26, and 27 show the worm detection rate for mixed traffic (worm instances & normal traffic). The X-axis indicates number of worm instances and normal traffic, Y-axis indicates the worm detection rate. The detection rate increases as the number of instances increases.

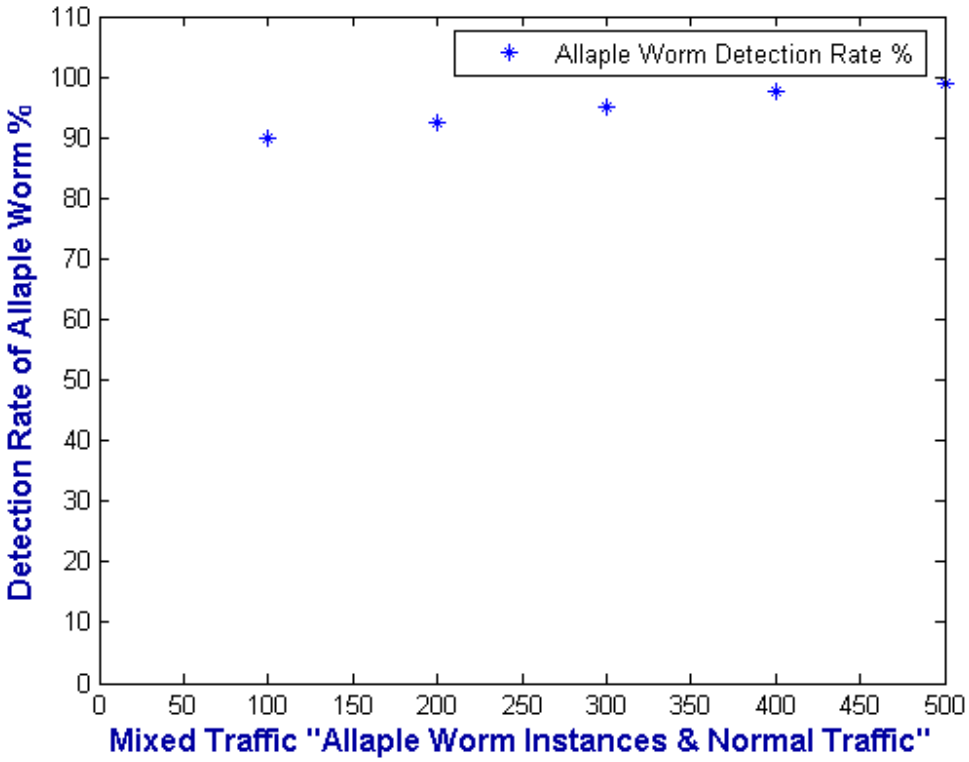

Figure 24. Allaple Worm Detection Rate

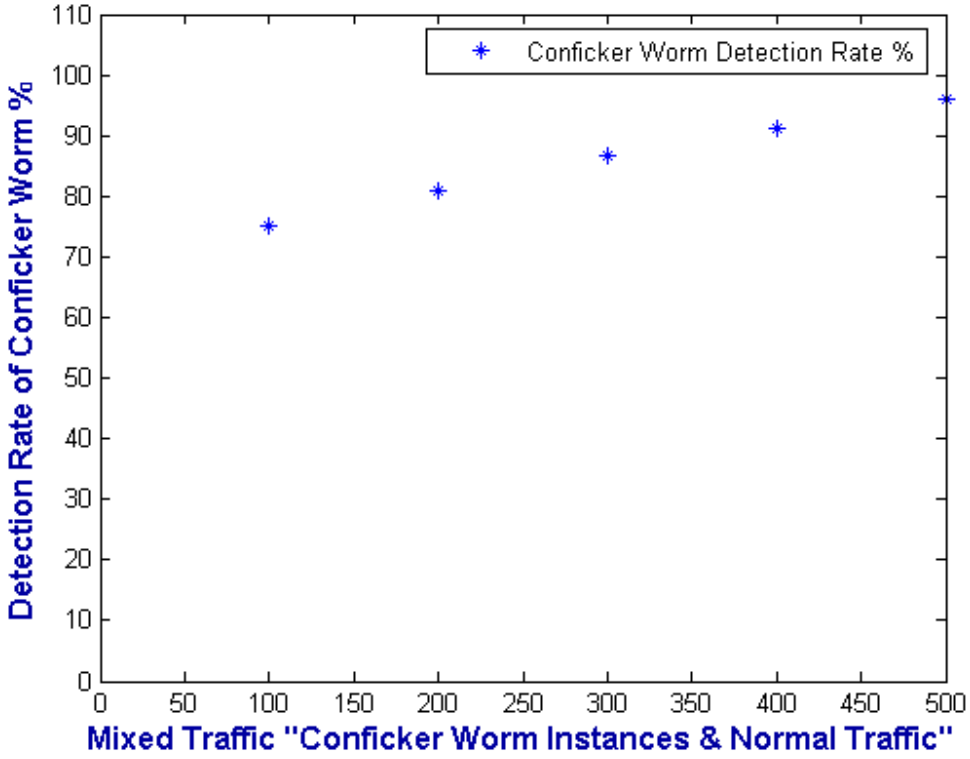

Figure 25. Conficker Worm Detection Rate

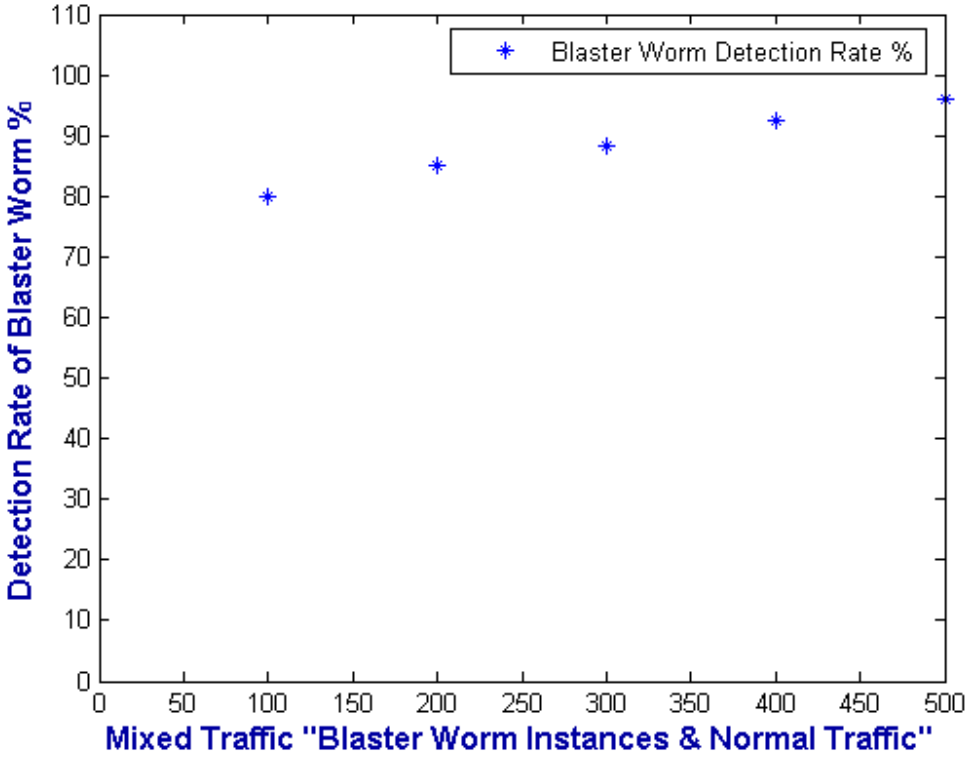

Figure 26. Blaster Worm Detection Rate

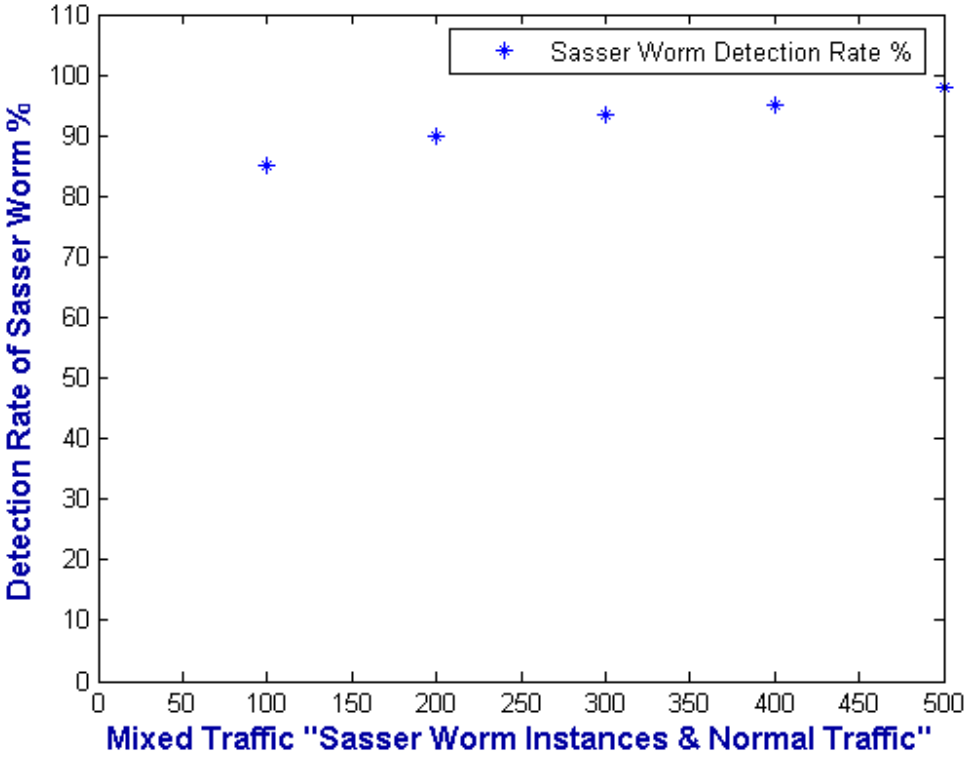

Figure 27. Sasser Worm Detection Rate

5.7.1.4 False Positives and Negatives Percentages

In the following, we calculate the false positives and false negatives for the four worms.

Figures 28, 29, 30, and 31 show the false positives and false negatives percentages for the four worms. We get zero false positives whereas the false negatives decrease as the number of worm instances increases.

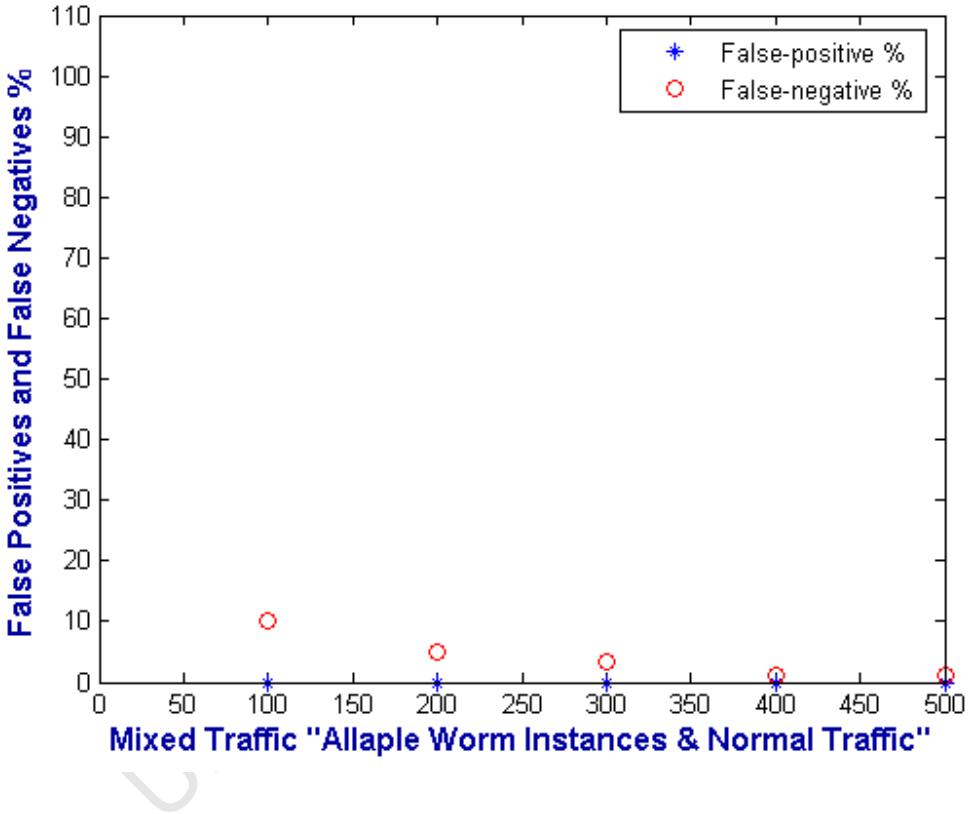

Figure 28. Allaple Worm False Positives and False Negatives

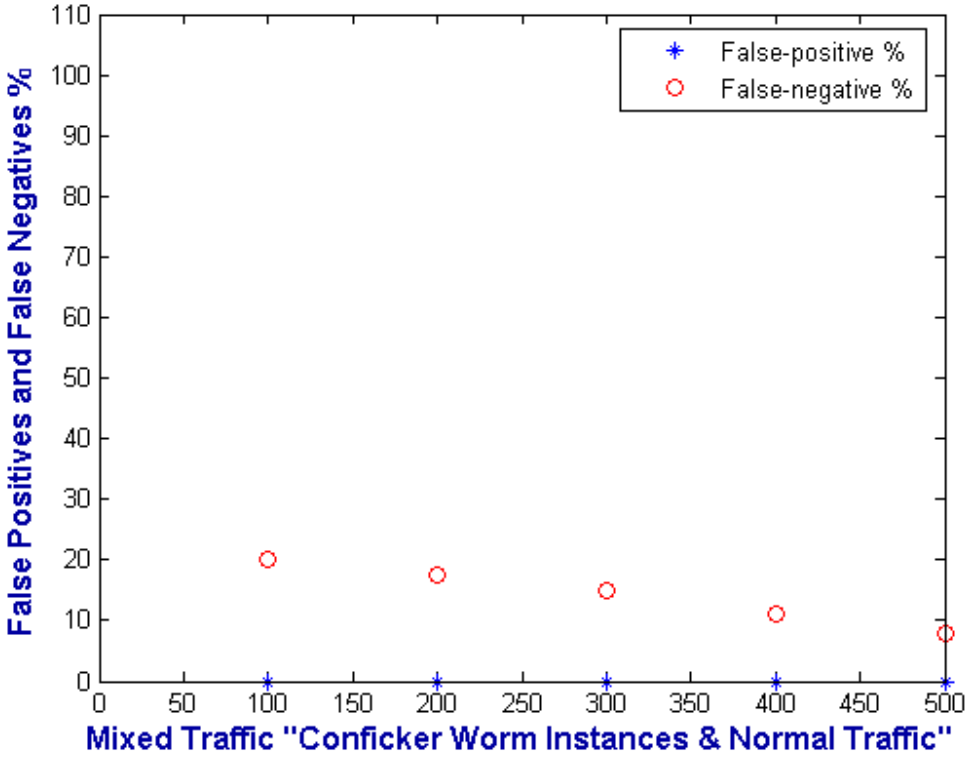

Figure 29. Conficker Worm False Positives and False Negatives

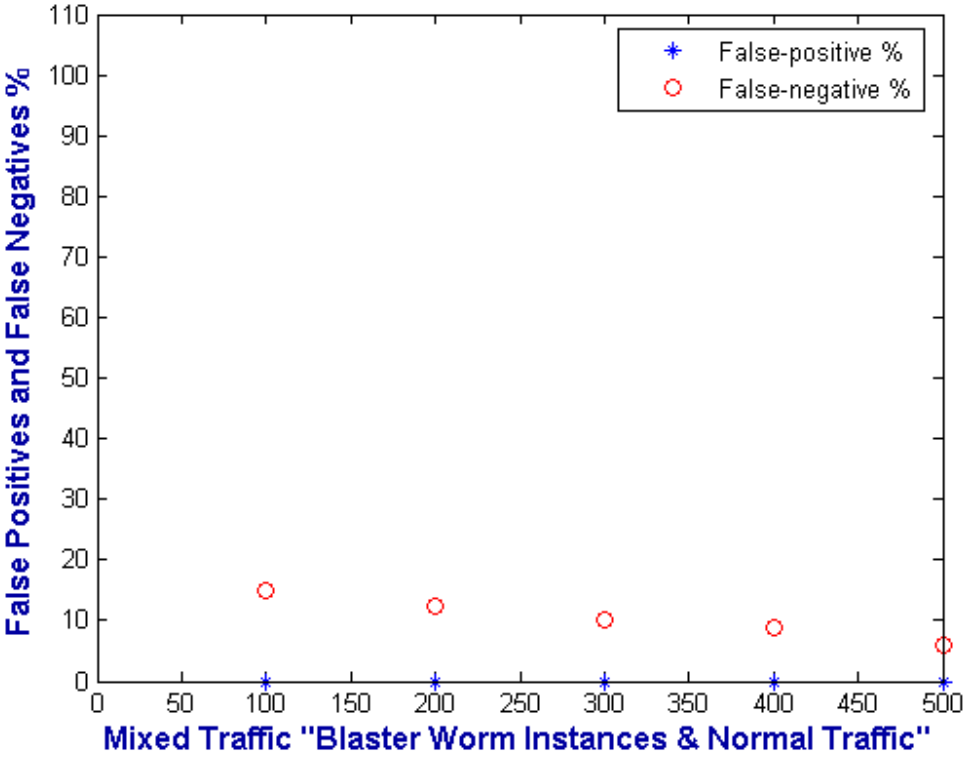

Figure 30. Blaster Worm False Positives and False Negatives

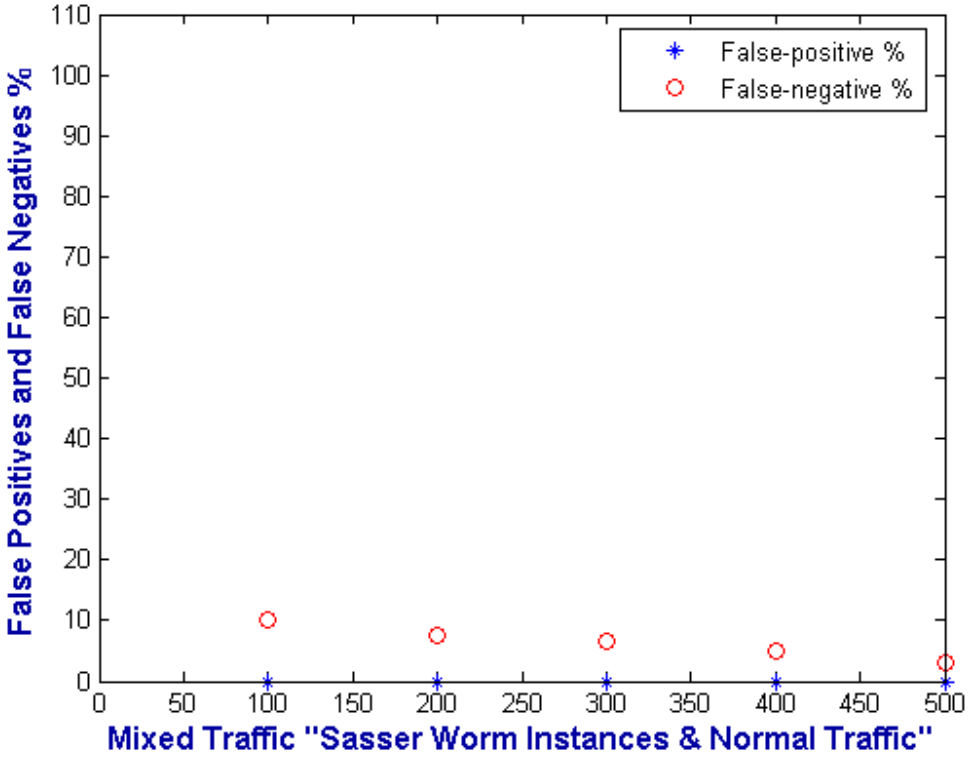

Figure 31. Sasser Worm False Positives and False Negatives

5.8 Clustering Method for Different Types of Polymorphic Worms

5.8.1 Euclidean distance

When our network receives different types of polymorphic worms (Mixed polymorphic worms), we must first separate them into clusters. As we mentioned in Section (5.4) we used the Euclidean distance to measure similarity in the nearest neighbor method. After the clustering is performed, we generate worm signature for each cluster in the same manner as we did in Subsection (5.2.1, 5.2.2, and 5.3.2).

The purpose of this experiment is to evaluate the effectiveness of the Euclidean distance in solving the clustering problem. In this experiment, 100 instances of Conficker worm (C_{w1}, Cw_2,…, C_{w100}) and 100 instances of Allaple worm (A_{w1}, A_{w2},…, A_{w100}) are used. The Euclidean distance is used to separate the mixed 200 instances into clusters.

The steps of the worms (Conficker and Allaple worms) separation process are described below:

1. Randomly we select one instance of Conficker or Allaple worm instances. Let us assume that the selected instance is A_{w1}.

2. We extract substrings form A_{w1} using the same manner explained in Subsection (5.2.1). Then we save the extracted substrings into an array, called W array.

3. We compute the frequency for each substring saved in W array in the remaining instances (A_{w2},…,A_{w100}, C_{w1},…, C_{w100}). Then we save the frequencies of each instance in a separate array.

4. We apply Euclidean distance to determine the similarity between the Conficker and Allaple worm instances according to their frequencies that computed in step 3.

Figure 32 shows the clustering problem between two mixed polymorphic worms (Conficker worm and Allaple worm). The X-axis indicates the number of mixed polymorphic worm instances, and y-axis indicates the similarity values between the mixed polymorphic worm instances. By using the Euclidean distance, the 200 worm instances are separated into two clusters, one for Conficker worm and one for Allaple worm.

As shown in Figure 32, we notice that the instances of the two worms lie in two different bands. To find the threshold between the two bands, we suppose that the band containing the Conficker worm instances lie above the band that contains the Allaple worm instances. Then we determine both the minimum value (denote it by minConficker) of the upper band and the maximum value of the lower band (denote it maxAllaple). The threshold is then given by:

$$\text{Threshold} = (\text{minConficker}+\text{maxAllaple})/2.$$

With the same above way, the Euclidean distance can cluster more than two mixed polymorphic worms.

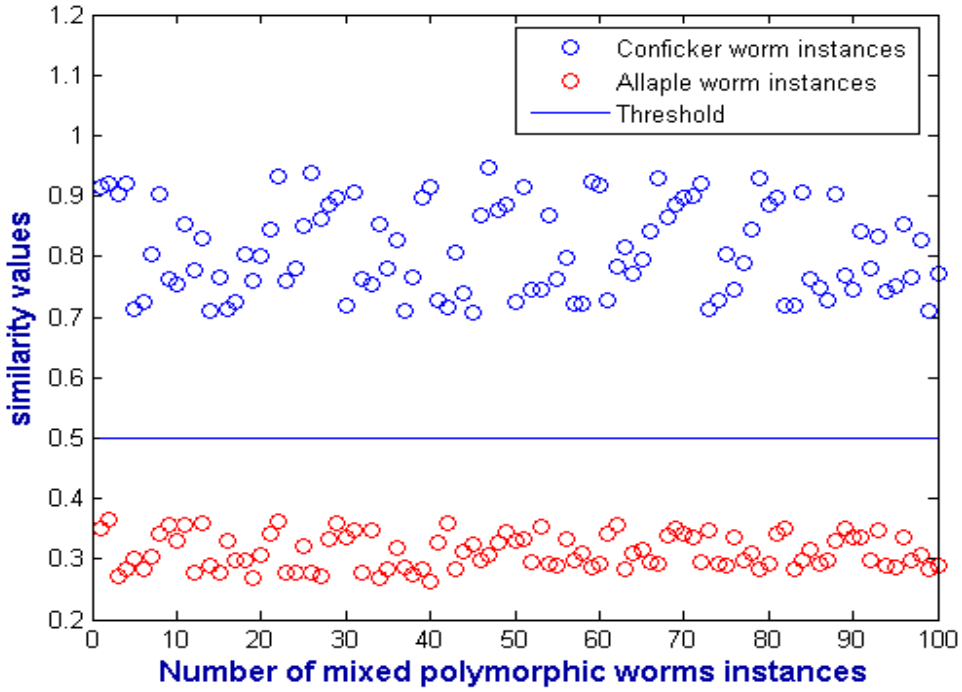

Figure 32. Clustering using Euclidean Distance

5.8.2 Clustering and Signature Generation Process example

Now, let us explain how our algorithms deal with mixed polymorphic worms environment. First, we separate the mixed polymorphic worms into clusters, and then we generate signatures for each cluster using the MKMP algorithm, and MPCA. Figure 33 shows clustering and signature generation processes for two mixed polymorphic worms.

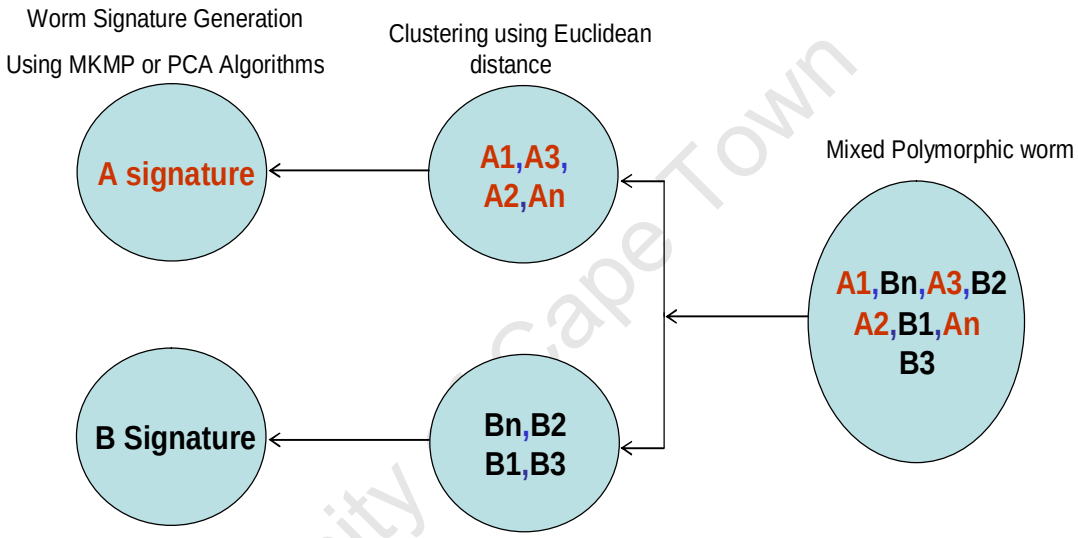

Figure 33. Clustering and Signature Generation for Mixed Polymorphic Worms

5.9 Comparison between Our System and the Existing Systems

In chapter 3, we gave accurate details about how our system outperforms the current automated signature generation systems for polymorphic worm. In addition to that discussion, the following table shows a comparison between our system and the currently existing automated signature generation systems for polymorphic worms.

Table 3. Comparative features of Our Systems and Existing Systems

	False Positives	False Negatives	Content/ behavior based	Signature Generation Algorithm	Single / Multiple Signature	Network/ Host based	Can the system collect most of the polymorphic worm instances?
Our System	Zero	Low	Content based	String Matching and Statistical algorithms	multiple substrings Signature	Both	Yes
Honeycomb [7]	Low	Low	Content based	String Matching	Single Substring Signature	Network based	No
Autograph [8]	Low	Low	Content based	String Matching	Single Substring Signature	Network based	No
Earlybird [9]	Low	Low	Content based	String Matching	Single Substring Signature	Network based	No
Hamsa [13]	Low	Low	Content based	Greedy approach	multiple substrings Signature	Network	No
LISABETH [14]	Low	Low	Content base	Greedy approach	multiple substrings Signature	Network	No

5.10 Chapter Summary

The first part of this chapter discussed algorithms which are used to generate signatures for polymorphic worms. Substring Extraction algorithm (SEA) is used to extract substrings from one of the polymorphic worm instances. The chapter described a modified version of KMP algorithm named MKMP algorithm. The MKPM is a signature generator algorithm that searches the occurrence of different words (extracted substrings) on string text (remaining instances). MPCA is a signature generator statistical approach that is used to reduce dimension of worm payload, so that the most significant data appear and are used as worm signature. Euclidean distance has been used to solve the clustering problem. The second part showed experimental performance results and analysis for the signature generation algorithms which are Modified Knuth–Morris–Pratt algorithm (MKMP algorithm) and Modified Principal Component Analysis (MPCA) in identifying polymorphic worms. The experimental results showed that the MKMP algorithm and the MPCA successfully detected polymorphic worms with zero false positives and low false negatives. The MKMP algorithm proved to be better because it received less false negatives than MPCA. The clustering method showed zero false negative.

Chapter 6 Conclusion and Recommendations for Future Work

The polymorphic worms evade signature-based Intrusion Detection Systems (IDSs) by changing their payloads in every infection attempt. This thesis designs a method for the detection of polymorphic worms attacks. The detection mechanism is based on two different approaches. In the first approach, a Double-honeynet system is proposed to collect all polymorphic worms instances. In the second approach, signatures are generated for the polymorphic worms instances that are collected by the Double-honeynet system.

The Double-honeynet system successfully collected different polymorphic worm instances as shown in Table 1. Then, two different methods, Modified Knuth-Morris-Pratt algorithm (MKMP algorithm) and the Modified Principal Component Analysis (MPCA), have been used to generate signatures for the polymorphic worms.

The MKMP algorithm compares the polymorphic worm substrings to find multiple invariant substrings which are shared among all polymorphic worm instances and are therefore used as the signatures of the polymorphic worm.

The Modified Principal Component Analysis (MPCA) is used to determine the most significant data which are shared among all the polymorphic worm instances and are then used as the signatures.

To test the efficiency of the proposed methods, malicious payloads of Conficker worm, Allaple worm, Blaster worm, and Sasser worm are used in the experiments. 200 instances of each of the above worms are used in this experiment, and 100 instances from each of the worms are used for signature generation. The rest of the instances are mixed with normal traffic to test the quality of the signature of each worm.

The experimental results show that the MKMP algorithm and the MPCA have successfully detected polymorphic worms with zero false positives and low false negatives.

It is recommended that for any real-life deployment of our system, real physical machines should be used with a multitude of different honeypots (running different operating systems and software applications) that will give the overall system the maximum ability to detect new zero-day polymorphic worms.

For future work, the system may be improved by using Artificial Intelligent (AI) algorithms such as neural networks and genetic algorithms.

References

[1] L. Spitzner, “Honeypots: Tracking Hackers,” Addison Wesley Pearson Education: Boston, 2002.

[2] M. Erbschloe, “Trojans, Worms, and Spyware: A Computer Security Professional’s Guide to Malicious Code, “Elsevier Butterworth–Heinemann 2005, 200 Wheeler Road, Burlington, MA 01803, USA.

[3] Hossein Bidgoli, “Handbook of Information Security,” John Wiley & Sons, Inc., Hoboken, New Jersey.

[4] S. G Cheetancheri, Collaborative, ”Defense Against Zero-Day and Polymorphic Worms: Detection, Response and an Evaluation Framework, ” PhD thesis, 2007. University of California, Davis. http://www.cs.ucdavis.edu/research/tech-reports/2007/CSE-2007-38.pdf

[5] D. Gusfield, “Algorithms on Strings, Trees and Sequences,”, Cambridge University Press: Cambridge, 1997.

[6] J. Levine, R. La Bella, H. Owen, D. Contis ,and B. Culver, "The use of honeynets to detect exploited systems across large enterprise networks," Proc. of 2003 IEEE Workshops on Information Assurance, New York, Jun. 2003, pp. 92- 99.

[7] C. Kreibich and J. Crowcroft, "Honeycomb–creating intrusion detection signatures using honeypots," Workshop on Hot Topics in Networks (Hot-nets-II), Cambridge, Massachusetts, Nov. 2003.

[8] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," Proc. of 13 USENIX Security Symposium, San Diego, CA, Aug., 2004.

[9] S. Singh, C. Estan, G. Varghese, and S. Savage, “Automated worm fingerprinting," Proc. Of the 6th conference on Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2004.

[10] James Newsome, Brad Karp, and Dawn Song," Polygraph: Automatically generating signatures for polymorphic worms," Proc. of the 2005 IEEE Symposium on Security and Privacy, pp. 226 – 241, May 2005.

[11] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," Proc. of the 14th conference on USENIX Security Symposium, 2005.

[12] Yong Tang, Shigang Chen," An Automated Signature-Based Approach against Polymorphic Internet Worms," IEEE Transaction on Parallel and Distributed Systems, pp. 879-892 July 2007.

[13] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao and Brian Chavez. Hamsa, “Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience,”

Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2006.

[14] Lorenzo Cavallaro, Andrea Lanzi, Luca Mayer, and Mattia Monga, “LISABETH: Automated Content-Based Signature Generator for Zero-day Polymorphic Worms,” Proc. of the fourth international workshop on Soft-ware engineering for secure systems, Leipzig, Germany, May 2008.

[15] Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, "Fast Automated Signature Generation for Polymorphic Worms Using Double-Honeynet," Proceedings of 3rd International Conference on Broadband Communications, Information Technology & Biomedical Applications (BroadCom 2008), Pretoria, South Africa, 23-26 November 2008.

[16] C. B. Moler, “Numerical Computing with Matalb, “Society for Industrial Mathematics, January 2004.

[17] Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," In ACM CCS, Nov. 2005. 98

[18] Z. Liang and R. Sekar, "Automatic generation of buffer overflow signatures: An approach based on program behavior models," In ACSAC, Dec. 2005. 98

[19] The HOL System, http://www.cl.cam.ac.uk/Research/HVG/HOL/, 3 january 2011.

[20] Mohssen Mohammed and H. Anthony Chan, "Honeycyber: Automated Signature Generation for Zero-day Polymorphic Worms," accepted at IEEE Military Communications Conference (MILCOM), San Diego, USA, 17-19 Nov. 2008. ISBN: 978-1-4244-2677-5.

[21] D. M. Ritchie,"The development of the C language," ACM SIGPLAN Notices, 28(3):201-208, 1993.

[22] D. Evans, "Static detection of dynamic memory errors," In SIGPLAN Conference on Programming Language Design and Im-plementation, PLDI'96, Philadelphia, PA, May 1996.

[23] D. Larochelle and D. Evans, "Statically detecting likely buffoverflow vulnerabilities," In Proceedings of the 10th USENIX Se-curity Symposium, pages 177-190, August 2001.

[24] H. Chen, D. Dean, and D. Wagner," Model checking one million lines of c code," In Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb 2004.

[25] H. Chen and D. Wagner," Mops: an infrastructure for examining security properties of software," In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS) , pages 235-244, Washington, DC, November 2002.

[26] D. Engler, B. Chelf, A. Chou, and S. Hallem," Checking system rules using system-specific, programmer-written compiler exten-sions," In Proceedings of the Fourth Symposium on Operating Sys-tems Design and Implementation, San Diego, CA, October 2000.

[27] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf," Bugs as deviant behavior: a general approach to inferring errors in sys-tems code," In SOSP '01 Proceedings of the eighteenth ACM sym-posium on Operating systems principles, pages 57-72 New York, NY, USA, 2001. ACM Press

[28] T. Ball and S. K. Rajamani,"Automatically validating temporal safety properties of interfaces," Lecture Notes in Computer Sci-ence, vol. 2057, May 2001, pp. 103-122.

[29] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang," Cyclone: A safe dialect of c," Proceedings of the 2002 USENIX Annual Technical Conference, Monterey, CA, USA, June 2002.

[30] G. C. Necula, S. McPeak, and W. Weimer," Ccured: type-safe retrotting of legacy code," In POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of program-ming languages, pages 128-139, New York, NY, USA, 2002. ACM Press.

[31] C. Cowan et al,"StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks," In Proc. of the 7th Usenix Security Symposium, pages 63-78, San Antonio, Texas, Jan 1998.

[32] R. Hastings and B. Joyce," Purify: Fast detection of memory leaks and access errors," In Proc. of the Winter 1992 USENIX Confer-ence, pages 125-138, San Francisco, California, 1991.

[33] B. Lampson," Dynamic protection systems," In Proceedings of the 35th AFIPS conference, pages 27-38, 1969.

[34] B. Lampson," Protection," In Proceedings of the 5th Annual Prin-ceton Conference on Information Sciences and Systems, pages 437-443, Princeton University, 1971.

[35] Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "A modified Knuth-Morris-Pratt Algorithm for Zero-day Polymorphic Worms Detection," Proceedings of The 2009 International Conference on Security and Management (SAM'09), Las Vegas, USA, 13-16 July 2009. IEEE. ISBN 9781601321244.

[36] S. Garfinkel and G. Spafford," Practical UNIX and Internet Secu-rity," O'Reilly,second edition, 1996.

[37] T. F. Lunt," Automated audit trail analysis and intrusion detection: A survey," In Proceedings of the 11th National Computer Security Conference, Baltimore, MD, 1988.

[38] Netfilter: Firewalling, nat and packet mangling for linux, http: //www.netfilter.org, 3, January 2011.

[39] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan. Buttercup, "On network-based detection of polymorphic buffer overflow vulnerabilities," In NOMS, Apr. 2004. 97

[40] M. Costa et al, "Vigilante: end-to-end containment of internet worms," In SOSP '05: Proceedings of the twentieth ACM Sympo-sium on Operating Systems Principles, pages 133{147. ACM Press, 2005.

[41] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on com-modity software," In NDSS, Feb. 2005.

[42] Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "Polymorphic Worm Detection Using Double-Honeynet," Proceedings of The Fourth International Conference on Software Engineering Advances (ICSEA 2009), Porto, Portugal, 20-25 September 2009. IEEE Computer Society. ISBN 9780769537771.

[43] L. Hui, "Color set size problem with applications to string matching," Proc. of the 3rd Symposium on Combinatorial Pattern Matching, Vol. 644, pp. 230 – 243, 1992.

[44] J. Nazario, "Defense and Detection Strategies against Internet Worms ", Artech House Publishers (October 2003).

[45] Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "A modified Knuth-Morris-Pratt Algorithm for Zero-day Polymorphic Worms Detection," Proceedings of The 2009 International Conference on Security and Management (SAM'09), Las Vegas, USA, 13-16 July 2009. IEEE. ISBN 9781601321244.

[46] C. C. Aggarwal and P. S. Yu, " Outliner Detection for High Dimensional Data," Proceedings of the ACM SIGMOD Conference, Santa Barbara, CA, May 21-24, 2001.

[47] Mohssen M Z E Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, and Eihab Bashier, "Fast and Accurate Detection for Polymorphic Worms," The 5th International Conference for Internet Technology and Secured Transactions (ICITST-2010),London, UK, 8.

[48] Vmware. http://www.vmware.com. 3 January 2011

[49] Know your enemy: Sebek, a kernel based data capture tool.http://www.honeynet.org/papers/sebek.pdf. 3 January 2011.

[50] Mohssen M. Z. E. Mohammed, H. Anthony Chan, Neco Ventura, Mohsin Hashim,Izzeldin Amin, "Accurate Signature Generation for Polymorphic Worms Using Principal Component Analysis," to appear in Proceedings of IEEE Globecom 2010 Workshop on Web and Pervasive Security (WPS 2010), Miami, Florida, USA,6-10 December 2010

[51] Know your enemy: Honeywall cdrom roo.http://www.honeynet.org/papers/cdrom/roo/index.html. 3 January 2011

[52] The Honeynet Project. Roo CDROM User's Manual, http://www.old.honeynet.org/tools/roo/manual/5-setup.html. 3 January 2011.

[53] Snort – The de facto Standard for Intrusion Detection/Prevention. Avail-able: http://www.snort.org, 3 January 2011.

[54] Bro Intrusion Detection System. Available: http://www.bro-ids.org/, 3, January 2011.

[55] C. Kruegel and G. Vigna, ”Anomaly Detection of Web-based Attacks, ” in Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS '2003). Washington D.C., USA: ACM Press, Oct. 2003, pp. 251-261.

[56] K. Wang and S. J. Stolfo, ”Anomalous Payload-based Network Intrusion Detection, ” in Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID '2004), Sophia Antipolis, French,Riviera, France, Sept. 2004.

[57] Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, and Izzeldin Amin, "Zero-day Polymophic Worms Detection Using a Modified Boyer-Moore Algorithm" Proceedings of the 2010 International Conference on Security and Management (SAM 2010), Las Vegas, USA, 12-15 July 2010.

[58] C. Kaufman, R. Perlman, and M. Speciner, ”Network Security: Private Communication in a Public World,” Upper Saddle River, New Jersey, USA: Prentice Hall, Inc., 2002.

[59] M. Christodorescu and S. Jha, ”Static Analysis of Executables to Detect Malicious Patterns, ” in Proceedings of the 12th USENIX Security Symposium (Security '2003), Washington D.C., USA, Aug. 2003, pp. 169-86.

[60] S. Staniford, V. Paxson, and N. Weaver,”How to 0wn the Internet in your spare time,” In USENIX Security Symposium, Aug. 2002.

[61] Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, And Izzeldin Amin, "Detection of Zero-day Polymorphic Worms using Principal Component Analysis", Proceedings of the Sixth International Conference on Networking and Services (ICNS 2010), Cancun, Mexico, 7-13 March 2010.

[62] CERT. Technical cyber security alerts, http://www.us-cert.gov.

[63] D. Moore, C. Shannon, G. Voelker, and S. Savage,” Internet quarantine: Requirements for containing self-propagating code,” In IEEE INFOCOM, Apr. 2003.

[64] N. Weaver, S. Staniford, and V. Paxson, ” Very fast containment of scanning worms, ” In USENIX Security Symposium, Aug. 2004.

[65] Mohssen M. Mohammed, H Anthony Chan, Neco Ventura, Mohsin Hashim, And Izzeldin Amin, " An Automated Signature Generation Approach for Polymorphic Worms Using Principal Component Analysis", the proceedings of the International Journal for Information Security Research (IJISR), Volume 1, Issue 1, ISSN: 2042- 4639 (Online).

[66] P. Fogla M. Sharif R. Perdisci O. Kolesnikov W. Lee. “Polymorphic Blending Attacks”; Proc. of the 15th conference on USENIX Security Symposium, Vancouver, B.C., Canada, 2006.

[67] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In 1996 IEEE Symposium on Security and Privacy: May 6{8, 1996,

Oakland, California, pages 190{200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.

Appendix A: Signature Generation Algorithms Pseudo Codes

In this appendix, we describe the signature generation algorithms pseudo codes. These algorithms were discussed in Chapter 5, and as we mentioned there, generating a signature for polymorphic worm A involves two steps:

- First, we generate the signature itself.
- Second, we test the quality of the generated signature by using a mixed traffic (new variants of polymorphic worm A and normal traffic).

1. Signature Generation Process

This section shows the Pseudo Codes for generating a signature for polymorphic worm A using SEA, MKMPA and MPCA.

1.1 Substring Extraction Algorithm (SEA) Pseudo Code

In the following we describe the pseudo code of the Substring Extraction Algorithm (SEA) that was discussed in [5.2.1]. The goal of SEA is to extract substrings from the first instance of polymorphic worm A and then to put them in an array W.

SEA pseudo code:

---

1. **Function** SubstringExtraction:
2. **Input** (a file A1: First instance of polymorphic worm A, x: minimum substring length)
3. **Output:** (W: array of substrings of A1 with a minimum substring length x)
4. **Define variables:**

Integer M : Length of file A1

Integer X: Maximum substring length

Integer z: (x<=z<=X) takes the lengths x to X

Integer Tz: Total number of substrings of file A1 with a substring length z

Integer position: the position of the first character of a substring of A1 with length z.

Array of characters S: a substring of A1 with length z

5. X= M-1

6. For z := x to X Do

7. Set Tz = M-z+1

8. Set position = 0

9. While position <= Tz

10. **S** = A1 (position) to A1(position+z-1)

11. Append (W, S)

12. position←position +1

13. EndWhile

EndFor

14. Return W.

1.2 Modified Knuth–Morris–Pratt (KMP) Algorithm Pseudo Code

In the following, we present the pseudo code for the Modified Kunth-Morris-Pratt algorithm (MKMP algorithm). Consider the example that we mentioned in Subsection (5.2.1) that we have a polymorphic worm with N instances (A_1, A_2,..., A_n). We select A_1 to be the instance from which we extract substrings. If G substrings are extracted from A_1, each substring will be equal to W_i for i= 1 to G. That means A_1 has G Words (W_1, W_2,..., W_G) whereas the remaining instances (A_2, A_3,..., A_n) are considered as S "text string".

The MKMP algorithm contains two functions:

a. *kmpfound* Function

kmpfound function is an MKMPA, which receives a word w from W array (W_1, W_2,…, W_G) and a File S (one file of the remaining instances A2,…,An) and determines whether w can be found in S or not.

b. S*ignatureFile* Function

SinatureFile function combined together with the above ***kmpfound*** function to get out the words (W_1, W_2,…, W_G) that appear in all of the remaining instances (A2,…, An), and use them as worm signature.

The MKMP algorithm has two inputs

- The first input is the substrings of the W array (the output of the SEA)
- The second input is the remaining instances (A2,…,An).

The goal of MKMP algorithm is to determine which substrings of W array appear in all remaining instances (A2,…,An), and to use them as a suspected worm signature.

MKMP Algorithm Pseudo code: *kmpfound* Function

1. **Function** *kmpfound*
2. **Inputs:**

   S: an instance of polymorphic worm A (A2, ..., An)

   w: a word from file W to be searched in file S /* W is the Output of the SEA */
3. **Output:**

   a boolean value (true if w is found in S, and false otherwise)
4. **Define variables:**

   an integer, m ← 0 (the beginning of the current match in S)

   an integer, i ← 0 (the position of the current character in w)

   an array of integers, T (the table, computed elsewhere)
5. **while** m+i is less than the length of S, **do**:
6. **if** w[i] = S[m + i],
7. **if** i equals the (length of w)-1,
8. **return** true
9. **let** i ← i + 1
10. **Otherwise,**
11. **let** m ← m + i - T[i],

```
12.          if T[i] is greater than -1,
13.              let i ← T[i]
14.          else
15.              let i ← 0
16.      Return false.
```

MKMP Algorithm Pseudo code: *SignatureFile* Function

1. **Function** SignatureFile
2. **Inputs:**

   W: Array of substrings of A1

   A2, ..., An: Instances of worm A
3. **Output:**

   SigFile : Array of substrings of A1 found in the rest instances (A2, ..., An) (Signature file contains the signature of the polymorphic worm A)
4. **Define variables:**

   FoundInAll: boolean variables which takes the value true if a word w(j) is found in all files A2, ..., An
5. **SigFile = Null**
6. **For** j := 1 To the length of W

```
7.        FoundInAll = True
8.        For k := 2 To n
9.            Use function KMPFound to check whether word W(j) can be found
          in file Ak
10.           If W(j) is not found in file Ak
11.               Set FoundInAll = False
12.           EndIf
13.       If FoundInAll
14.       Append W(j) to file SigFile
15.       EndIf
16.   EndFor
17. EndFor
18. Return SigFile
```


1.3 Modified Principal Component Analysis (MPCA) Pseudo Code

Here, we present the pseudo code of MPCA method which contains two functions:

a**. Compute *Array of Frequencies Function***, The goal of this function is to compute the frequencies of each substring in W array in the remaining instances (A2,..,An). W array contains the substrings extracted by the Substring Extraction Algorithm (SEA).

The inputs to this function are W array and the remaining instances (A2,…An). The

output of this function is the frequencies of each W substring in the remaining instances (A2,…An).

b. ***Compute Principal Component Function***, This function computes the most important components and uses them as worm signature.

The goal of this function is to extract the Feature Descriptor, which contains the most important features of polymorphic worm A.

The input to this function is matrix FFF which is the output of the ***Compute Array of Frequencies*** Function. The output of this function is the Feature Descriptor of polymorphic worm A.

In the following we describe the pseudo codes for ***Compute Array of Frequencies*** Function and ***Compute Principal Component*** Function.

Modified Principal Component Analysis (MPCA): Compute Array of Frequencies Function Pseudo code

1. **Function** ComputeArrayOfFrequencies
2. **Inputs:** (Instances A2,..,An, Array W)).
3. **Output** (Matrix FF of frequencies of substrings of A1 stored in array W in files A2,...,An), and a vector of integers Zr)
4. **Define variables**

   Integers: X,j,k, Wlength

   Matrices of Real: FF, FFF (FFF is the matrix will be obtained by reducing all the zero rows of matrix FF)
5. **Set** x= Minimum substring length

6. **W** := ***SubsringExtraction*** (A1, x)
7. **Wlength** := Length (W) (number of substrings extracted in W array)
8. **FF** = Matrix (Wlength, n-1) /* n is the number of polymorphic worm A instances */
9. **for** j from 1 To Wlength Do
10. **for** k from 1 to n-1 Do
11. **set** FF(j, k) be the frequency of word W(j) in file A(k+1)
12. **EndFor**
13. **EndFor**
14. Remove all zero rows from FF giving Matrix FFF of size Nx(n-1) and save indexes of zero rows in a vector Zr
15. **Return** FFF and Zr

Modified Principal Component Analysis (MPCA): Compute Principal Component Function Pseudo code:

---

1. **Function** ComputePrincipalComponents:
2. **Inputs**(FFF, K: Number of most important feature)
3. **Output** (FD: a matrix of feature descriptors)
4. **Define variables:**

Matrices of Real: D, G, C, evecs, evals, PC (**D**: matrix of normalized

frequencies; **G**: matrix of Mean Adjusted Data; **C**: covariance Matrix; **evecs**: matrix of eigenvectors of covariance Matrix; **evals**: matrix of eigenvalues of covariance matrix; **PC**: matrix consisting set of principal component vectors)

5. **FFF** = ComputeArrayofFrequnciesMatrix (A2,...An, W)
6. **FFFRows** = Number of rows of FFF
7. **FFFCols** = Number of columns of FFF
8. **Compute** the matrix of normalized frequencies D = (dij) using dik $\leftarrow \frac{f_{ik}}{\sum_{j=1}^{N} f_{ij}}$
9. **Set** $\bar{d}_i$ (mean of the ith row of D)
10. **Compute** matrix G = (gik) where gik= dik- $\bar{d}_i$
11. **Compute** the covriance matrix C (Ci,j) where Cij = $\frac{\sum_{k=1}^{L}(d_{ik} - \bar{d}_i)(d_{jk} - \bar{d}_j)}{N-1}$, (C is NxN matrix)
12. **Compute the eigenvalues of C** (λ1, λ2,..., λn) by solving $|C - \lambda I|$ = 0, sorted in a descending order of their magnitudes.
13. **Compute** the eignvetors of C V1, V2,..., Vn corresponding to the eigenvalues of C.
14. **Let** matrix V be the matrix whose columns are the eigenvectors vj^T (j=1,..,k)
15. **Compute** the Feature Descriptor FD = V^T x FFF
16. **Return** FD

Pseudo Codes for testing the quality of the generated signature for polymorphic worm A will be discussed in the following section.

2. Testing the Quality of the Generated Signature for Polymorphic Worm A

In this section, we show the MKMP algorithm and MPCA pseudo codes for testing the quality of the generated signature for polymorphic worm A (where this signature was generated in Section 1 by using SEA, KMP, and MPCA algorithms). To test the quality of the signature, we use a mixed traffic (new variants of polymorphic worm & normal traffic i.e. innocuous packets). The new variants of polymorphic worm A are not as the same as the variants that were used to generate the signature. (i.e. training set is A1,A2…An. Test set is An+1,….,Am, where $m > n$).

In the following we describe the pseudo codes of the MKMP algorithm and MPCA that we use to test the quality of the generated signature for polymorphic worm A.

Modified Knuth–Morris–Pratt (KMP) Algorithm Pseudo Code for testing the generated signature for polymorphic worm A

---

1. Inputs: a packet P (which can be suspicious (An+1,..,Am) or innocuous packet), and SigFile which contains the signature of Polymorphic Worm A that was generated using SignatureFile function)

2. Output: a boolean value (true if all substrings of SigFile are found in packet P, and false otherwise)

3. If ***kmpFound*** (P, SigFile)
Retrurn True
Otherwise
Return False.

MPCA Pseudo Code for testing the quality of the generated signature for polymorphic worm A

1. Inputs: a packet P (which can be suspicious (An+1,..,Am) or innocuous packet), W array, the vector Zr; and the Polymorphic worm A's Feature Descriptor(FD) and threshold r which was calculated using the ComputePrincipalComponents function)

2. Output: a boolean value (true if the Euclidean distance between the FD and Packet P <= r, and false otherwise)

3. Define Variable
Let k = number of rows of FD.

4. Use function ***FunctionComputeArrayOfFrequencies*** to compute the frequencies of substrings of W array in Packet P, save the frequencies in a vector Fj and remove components of Fj indexed by Zr (Dimension of Fj is as same as FD).
5. Calculate the Euclidean distance between rows of **FD** and **Fj** Then save it a matrix **Dt**.

6. If for some j (1<=j<=k) the distance Dt(j) is less than the threshold value r, return true, otherwise **return** False

Appendix B: Double-honeynet System Configurations

In the following we discuss the Double-honeynet system architecture, and configuration using Vmware.

1. Double-honeynet Architecture Implementation

In the following, we discuss the implementation of the Double-honeynet system on VMware. Figure 34, shows the architecture of the Double-honeynet system, implemented using VMware Workstation version 7 on PC Intel Pentium 4, 3.19-GHZ CPU, 8GB RAM, the PC running Windows XP 64-bit. The operating system of that personal computer is referred as host operating system in Figure 34. The host machine was connected to our home router and it accessed the Internet through it.

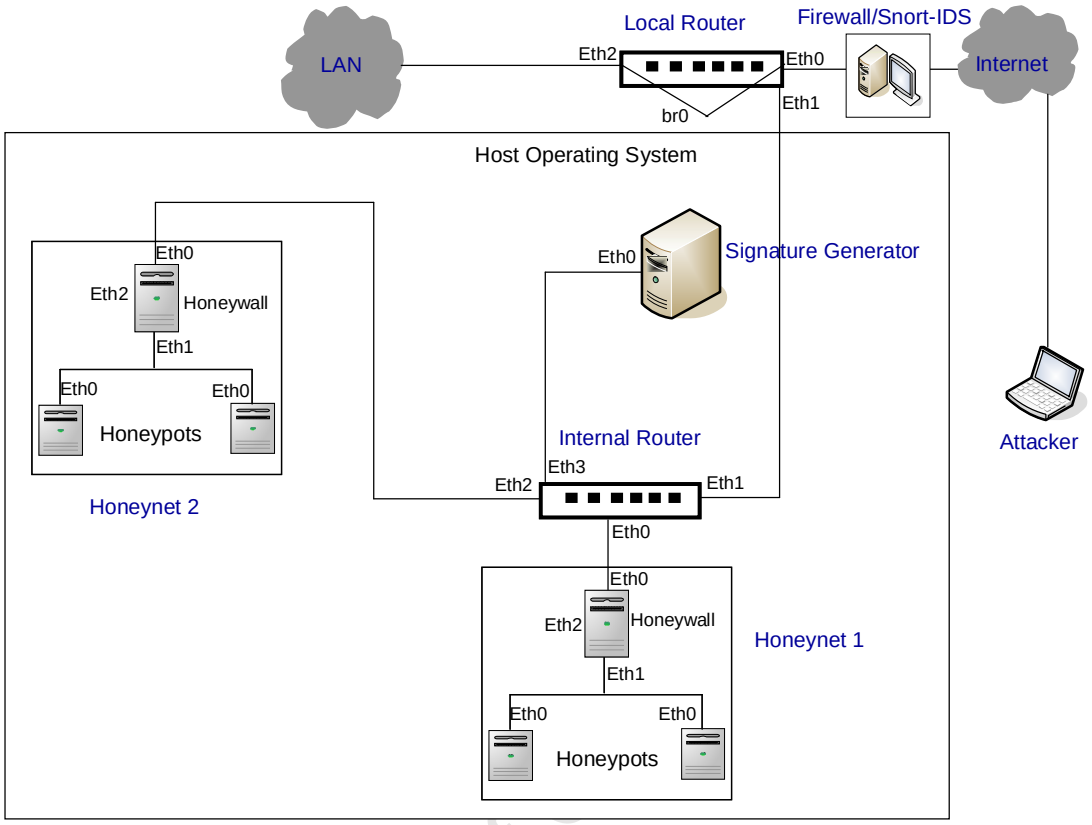

Figure 34. Double-honeynet Architecture

We used virtual machine to deploy the Double-honeynet system due to the lack of resources and keeping the establishment low-cost. One personal computer (PC) was used and VMware Workstation was installed on it. The VMware Workstation is a software package that gives its users the opportunity to create virtual machines that constitute virtual networks interconnected with each other [48]. Thus, we created the Double-honeynet system as a virtual network seen from the outside world as an independent network. Attackers could locate the Honeypot and attack it. The Honeypot was transparently connected to the Internet through the Honeywall which in turn intercepted all outbound and inbound traffic. Therefore, malicious traffic targeting the Honeypot (inbound) or malicious traffic generated by the compromised Honeypot (outbound) were available to us from the Honeywall for further analysis and examination. As we mentioned in (4.2), Honeynet 1 and Honeynet 2 were configured to deliver unlimited outbound connections. The *Internal Router* was used to protect our local network by redirecting all outbound connections from Honeynet 1 to Honeynet 2 and vice versa.

2. Configuration

In the following, we discuss the Double-honeynet configuration details.

Our Double-honeynet system contains six components which are Local Router, Internal Router, LAN, Honeynet 1, Honeynet 2, and Signature Generator. The subnet mask for each subnet (whether Local Router, Internal Router, LAN, Honeynet 1, Honeynet 2, and Signature Generator) is consequently 255.255.255.0. The following subsequent sections discuss the configurations of each component.

2.1 Local Router Configuration

As we mentioned in Section (4.2) Local Router's function is to pass unwanted traffic to the Honeynet 1 through the Internal Router. For example, if the IP address space of our LAN is 212.0.50.0/24, with one public Web server, the server's IP address is 212.0.50.19. If an attacker outside the network lunches a worm attack against 212.0.50.0/24, the worm scans the IP address space of victims. It is highly probable that an unused IP address, e.g. 212.0.50.10 will be attempted before 212.0.50.19. Therefore, Local Router will redirect the packet to Honeynet 1 through Internal Router. After the worm compromised the Honeynet 1, the worm will try to make an outbound connection to harm another network. We configured the Internal Router to protect the LAN from worms' outbound connections. The Internal Router intercepts all outbound connections from Honeynet 1 and redirects them to Honeynet 2 which performs the same task being done by the Honeynet 1 forming loop connections. Below are the details of the Local Router machine properties and IPtables configuration.

- **Machine Properties:**
    - Operating System: Ubuntu linux 9.10
    - Number of Network Cards:

Three network cards (Eth0, Eth1, and Eth2).

Eth0 and Eth2 are bridged LAN port.

Eth1 function is to connect Local Router with Honeynet 1 through the Internal Router.

- IP Addresses:
  - Eth1: 192.168.50.20
- Prior to the IPtables setting, we enabled IP forwarding in the Local Router.

  -Edit /etc/sysctl .conf file as follows :

  Net.ipv4.ip_frowrd =1
- IP-tables configuration

  The settings of the Network Address Translator (NAT) in the kernel using IPtables are as follows:

  1. Do not translate packets going to the real public server:

     # iptables -t nat -A PREROUTING -m physdev --physdev-in eth0 -d 212.0.50.19 -j RETURN

  2. Translate all other packets going to the public LAN to Internal Router:

     # iptables -t nat -A PREROUTING -m physdev --physdev-in eth0 -d 212.0.50.0/24 -j DNAT --to 192.168.50.22

2.2 Internal Router Configuration

As we mentioned in Section (4.2), the Internal Router's function is to protect the LAN from worms' outbound connections and to redirect the outbound connections from Honeynet 1 to Honeynet 2 and vice versa. Let us investigate more about the Internal Router machine properties and IPtables configuration in the following texts.

- **Machine Properties**
  - Operating System : Ubuntu linux 9.10
  - Number of Network Card:

    Four network cards (Eth0, Eth1, Eth2, and Eth3).

    Eth0 function is to connect Internal Router to the Honeynet1-clients.

    Eth1 function is to connect Internal Router with Local Router.

    Eth2 function is to connect Internal Router with Honeynet 2-clients.

    Eth3 function is to connect Internal Router with Signature generator.

  - **IP Addresses:**

    Eth0: 192.168.51.20

    Eth1: 192.168.50.22

    Eth2: 192.168.58.20

    Eth3 192.168.55.20

  - Before we set the Iptables rules we enable the IP Forwarding in Internal Router:

    -Edit /etc/sysctl .conf file as follows :

    Net.ipv4.ip_frowrd =1

  - **IPtables configuration:**

    The settings of the Network Address Translator (NAT) in the kernel using IPtables are as follows:

    1. Translate packets coming in from eth1 to the Honeynet 1

```
# iptables -t nat -A PREROUTING -i eth1 -j DNAT --to 192.168.51.22
```

2. From Honeynet 1 don't translate packets to the signature generator

```
# iptables -t nat -A PREROUTING -i eth0 -s 192.168.51.22 -d 192.168.55.22 -j RETURN
```

3. From Honeynet 1 translate all other packet to Honeynet 2

```
# iptables -t nat -A PREROUTING -i eth0 -j DNAT --to 192.168.58.22
```

4. From Honeynet 2 don't translate packets to the Signature generator

```
# iptables -t nat -A PREROUTING -i eth0 -s 192.168.58.22 -d 192.168.55.22 -j RETURN
```

5. From Honeynet 2 translate all other packet to Honeynet 1

```
# iptables -t nat -A PREROUTING -i eth0 -j DNAT --to 192.168.51.22
```


2.3 LAN Configuration

As described in Subjection (2.1), we have one public Web server in our LAN with this IP address: 212.0.50.19. Below are the details of the public web server machine properties.

- **Machine Properties:**
  - Operating System: Ubuntu linux 9.10
  - *Number of Network Card:*

    One network card Eth0.
  - **IP Address:**

    Eth0: 212.0.50.19.

2.4 Honeynet 1

As shown in Figure 34, Honeynet 1 contains Honeywall and two Honeypots. The main function of the Honeynet 1 is to capture polymorphic worms instances. Below are the details of the Honeywall machine properties and configuration.

- **Machine properties:**
  - Number of Network Cards:

    Three network cards (Eth0, Eth1, and Eth2).

    Eth0 function is to connect Honeynet 1 with Honeynet 2 through Internal Router.

    Eth1 function is to connect Honeynet 1 with his clients (Honeypots).

    Eth2: used for Management interface.
  - **IP Addresses:**

    Eth0: 192.168.51.22

    Eth1: 192.168.52.20

    Eth2: 192.168.40.7
  - **Honeywal configurations**

    1. *Honeynet public IP Addresses*

       In the following, we type the external IP addresses for the honeypots. These are the IP addresses which are attackers:

       IP addresses: 192.168.52.22 192.168.52.23

    2. *Honeynet Network*

       In the following, we type the Honeynet network in CIDR (Classless Inter-

Domain Routing) notation:

Honetnet Network CIDR: 192.168.52.0/24

3. *Broadcast address of the Honeynet* : 192.168.52.255

4. *Management Interface:*

Third interface will be used for remote management. This interface helps us to remotely manage the Honeywall through SSH and Walleye Web interfaced. We use Eth2 for the management interface.

IP address of the Management interface:192.168.40.7

Network mask of the management interface: 255.255.255.0

Default gateway for the management interface:198.168.40.1

DNS server IP for honeywall gateway :192.168.40.2

SSHD listening port: 22

Space delimited list of TCP ports allowed into the management interface:22 443

Space delimited list of IP addresses that can access the management interface:192.168.40.0/24

5. *Firewall Restrictions:*

The Double-Honeynet configured to perform unlimited outbound connections as mentioned in (5.2) above.

6. *Configure Sebek Variables*

Sebek is a data capture tool designed to capture the attackers' activities on a honeypot. It has two components. The first is a client that runs on the

honeypots; its purpose is to capture all of the attackers' activities (keystrokes, file uploads, passwords), then covertly to send the data to the server. The second component is the server which collects the data from the honeypots. The server normally runs on the Honeywall gateway.

Destination IP address of the sebek packets:192.268.52.20

*Destination UDP port of the Sebek Packets:*1101

7. *Honeypots Configuration*

Below are the details of the Honeypots machines properties and configuration.

- **Honeypot 1**
  - **Machine properties:**
    - Operating System: Windows XP
    - Number of Network Card:

      We use one network card Eth0.
    - IP Address:

      Eth0: 192.168.52.22
- **Honeypot 2**
  - **Machine properties:**
    - Operating System: Ubuntu linux 9.10
    - Number of Network Card:

      We use one network card Eth0.
    - IP Address:

Eth0: 192.168.52.23

2.5 Honeynet 2 Configuration

Honeynet 2 contains Honeywall and two Honeypots. Honeynet 2 function is to capture polymorphic worms instances. Below are the details of the Honeywall machine properties and configuration.

- **Machine properties:**
  - **Number of Network Cards:**

    Three network cards (Eth0, Eth1, and Eth2).

    Eth0 function is to connect Honeynet 2 with Honeynet 2 through Internal Router.

    Eth1 function is to connect Honeynet 2 with his clients (Honeypots).

    Eth2 used for Management interface.

  - **IP Addresses:**

    Eth0: 192.168.58.22

    Eth1: 192.168.59.20

    Eth2: 192.168.40.8

  - **Honeywall configuration**

    1. *Honeynet public IP Addresses*

       In the following, we type the external IP addresses for the honeypots. These are the IP addresses which are attackers:

       IP addresses: 192.168.59.22 192.168.59.23

    2. *Honeynet Network*

In the following, we type the Honeynet network in CIDR (Classless Inter-Domain Routing) notation:

Honetnet Network CIDR: 192.168.59.0/24

3. *Broadcast address of the Honeynet* : 192.168.59.255

4. *Management Interface:*

Third interface will be used for remote management. This interface helps us to remotely manage the Honeywall through SSH and Walleye Web interfaced. We use Eth2 for the management interface.

IP address of the Management interface:192.168.40.8

Network mask of the management interface: 255.255.255.0

Default gateway for the management interface:198.168.40.1

DNS server IP for honeywall gateway :192.168.40.2

SSHD listening port: 22

Space delimited list of TCP ports allowed into the management interface:22 443

Space delimited list of IP addresses that can access the management interface:192.168.40.0/24

5. *Firewall Restrictions:*

The Double-Honeynet configured to perform unlimited outbound connections as mentioned in (5.2) above.

6. *Configure Sebek Variables*

Destination IP address of the sebek packets:192.68.59.20

*Destination UDP port of the Sebek Packets:*1101

7. *Honeypots Configuration*

Below are the details of the Honeypots machines properties.

- **Honeypot 1**
  - **Machine properties:**
    - Operating System: Windows XP
    - *Number of Network Card:*

      We use one network card Eth0.
    - IP Address:192.168.59.22
- **Honeypot 2**
  - **Machine properties:**
    - Operating System: Ubuntu linux 9.10
    - Number of Network Card:

      We use one network card.
    - IP Address: 192.168.59.23

2.6 Signature Generator Configuration

The function of the signature generator is to generate signatures for polymorphic worms instances which were collected by the Double-honeynet system using algorithms that were discussed in Chapter (5).

- **Machine Properties:**
    - Operating System: Ubuntu linux 9.10
    - Number of network cards:

      One network card Eth0.
    - IP address:

      Eth0: 192.168.55.22